Diss. ETH No:

MAN-MACHINE INTERFACES AND IMPLEMENTATIONAL ISSUSES IN COMPUTER-AIDED CONTROL SYSTEM DESIGN

A DISSERTATION

submitted to the
SWISS FEDERAL INSTITUTE
of
TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
CARL MAGNUS RIMVALL
Civilingenjör LTH, Lund, Sweden
born February 24, 1957
Swedish citizen

accepted on the recommendation of
Prof. Dr. M. Mansour, referee
Prof. Dr. F.E. Cellier, co-referee

Zurich 1986

© Copyright Magnus Rimvall, 1986
All rights reserved.

Contents

List of Figures

Acknowledgements

I wish to express my sincere gratitude to Professor Mohammed Mansour for giving me the opportunity to conduct the research presented in this thesis at the Department of Automatic Control of ETH. I am especially thankful for the help stemming from his farsightedness of the general trends in this research area and for his generous provision of travel funds, which has enabled me to establish many invaluable, international contacts. Furthermore, I am greatly indebted to Professor François Cellier, University of Arizona, Tucson, for introducing me to this field of research, for agreeing to act as co-referee of this thesis, and for everything else he has done for me. I am particularly grateful for the frequent and most rewarding scientific discussions during his time at the ETH, and for our unfortunately more infrequent, but still very rewarding, discussions and collaborations since his relocation to Arizona.

Furthermore, I would like to express my thanks to all colleagues in the field of CACSD around the world with whom I have collaborated. Special thanks goes to Professor Karl-Johan Åström, Sweden, Dr. Jan Maciejowski, England, Professor Dean Frederick, USA, Dr. Charlie Herget, USA, and all the members of their respective research groups. I also appreciate many fruitful discussions with Kent Diehl, Dr. Michael Floyd, John Little, Brad Schrick, Dr. Steven Schladover, Dr. Peter Thompson, and other developers of commercial CACSD products.

Credit must also be given to my co-workers at the Department of Automatic Control and to the various students who have assisted me during Semester- and Diploma-projects in the development of the software described in this thesis. Among these students, special credit must be given to Lars Bomhold for the implementation of the basic data structures and driving programs of the Query-feature, and to Fredi Schmid for the development of the nonlinear system description syntax and parser.

I finally want to thank everybody else who in any way have contributed to the completion of this thesis. I would like to thank Franz Kuster for forsaking his laser-printer during several weeks at the end of this project, and for his willing assistance by all technical problems. A special thanks goes to my friend Selim Sedat Hacısalihzade, with whom I have shared office the last four years. I enjoyed our discussions over a broad range of topics and appreciate his decision to act as a sparring partner for the completion of our respective research projects. I was also delighted to win this competition with a margin of 45 minutes. Last, but not least, I would like to thank my wife, Dr. Karin Rimvall-Vallin, for her patience and her practical assistance in the end phase of this work. However, with Karin there was no competition — I lost with over half a year...

Summary

This thesis deals with different issues of **Computer-Aided Control System Design (CACSD)** software from a control engineering and computer engineering viewpoint. Emphasis is given to the design of user interfaces to interactive CACSD packages, to the control-oriented data structures to be supported by such packages, and to the software engineering problems of implementing large CACSD packages.

The design of a good user interface to CACSD packages is of utmost importance for user acceptance of the package and for the applicability of the software to of-the-shelf problems. Different approaches to the design of user interfaces are discussed and a complete, algorithmically extendable, command driven, interface is presented. This interface has been implemented in the *IMPACT* package as an integral part of the research project presented here and numerous examples from IMPACT illustrate different CACSD aspects throughout the thesis.

An obvious requirement of any CACSD package is that the program must be able to represent, manipulate and properly display the numerical, symbolic or logical entities which are needed during the control design cycle. However, this basic requirement was hitherto not necessarily fulfilled even in commercially successful CACSD packages. Therefore, this topic is discussed in detail, with special emphasis on assembling an adequate and yet perspicuous set of data structures. An unambiguous scheme for *overloading* mathematical operators, and thus to enhance the expression power of the command language manipulating on these data structures, is presented.

Despite recent and not-so-recent advances in the theory of structured programming and software engineering, there is a longstanding "tradition" to implement all scientific programs in

FORTRAN. This was hitherto partially due to a lack of viable alternatives. Hence, the aptitude of the Ada programming language as an alternative for implementing large interactive programs in general, and CACSD packages in particular, is investigated. Several implementational schemes for error handling, data management, command language interpreter, overall package design et cetera are presented to validate the suitability of Ada for this task.

CACSD is a multifaceted field and therefore this thesis involves issues from a large number of fields including control theory, software engineering, computer graphics and formal language theory. Since most readers will be specialists in only some of these areas, the background needed for understanding each section is to a certain extent provided and ample references to relative literature is also given. However, a working knowledge of basic control theory and some structured programming language is assumed.

Zusammenfassung

Diese Dissertation behandelt verschiedene Aspekte des **CACSD (Computer-Aided Control System Design** oder Rechnerunterstützter Regelungsentwurf) sowohl aus der Sicht der Regelungstechnik als auch von der Seite der Informatik und des Software Engineering. Die Schwerpunkte der Dissertation liegen im Entwurf von flexiblen Benützerschnittstellen zu interaktiven CACSD-Programmen, in der Erstellung einer Gruppe der Regelungstechnik angepassten Datenstrukturen, und bei der Problematik des Software-Engineering in der Implementation grösserer CACSD-Pakete.

Für einen erfolgreichen Einsatz eines CACSD Paketes bei seinem Benützer ist die Unterstützung einer an die Bedürfnisse der Regelungstechniker angepassten Benützerschnittstelle von ausserordentlicher Bedeutung. Insbesondere muss diese Benützerschnittstelle erweiterbar sein, so dass das Paket über die Behandlung von Standardprobleme hinaus einsetzbar bleibt. In der vorliegenden Arbeit werden verschiedenste Ansätze zum Entwurf solcher Benützerschnittstellen diskutiert und eine vollständige, algorithmisch erweiterbare, kommandosprachgesteuerte Benützerschnittstelle vorgestellt. Diese Schnittstelle wurde im Rahmen dieser Forschungsarbeit auch im CACSD-Paket IMPACT implementiert. Zur Veranschaulichung des Stoffes werden sämtliche Teile der Dissertation durch Beispiele von IMPACT illustriert.

Eine selbstverständliche Anforderung an alle CACSD-Pakete ist, dass sie die numerischen, symbolischen und/oder logischen Daten, die der Benützer während der Entwurfsarbeit braucht, auch abspeichern, verarbeiten und geeignet darstellen können. Leider ist diese Grundanforderung nicht einmal in den heutigen kommerziell erfolgreichsten Paketen erfüllt. In dieser Arbeit

wird deshalb die Problematik der Unterstützung hinreichender aber immer noch überschaubarer Datenstrukturen vorgestellt und diskutiert. Eine kompakte, "überladene" (overloaded) aber immer noch eindeutige algorithmische Notation, mit der die Ausdrückbarkeit der Eingabe gesteigert werden kann, wird vorgestellt.

Trotz allen wohlbekannten Theorien des strukturierten Programmierens und des Software-Engineering, gibt es nur wenige Ausnahmen zur allgemeinen Tradition, alle wissenschaftlichen Programme in FORTRAN zu implementieren. Dies war bisher, mindestens teilweise, auf mangelnde Alternativen zurückzuführen. In dieser Arbeit werden die Einsatzmöglichkeiten der Programmiersprache Ada für die Implementation grösserer interaktiver Programme im allgemeinen, und CACSD im besonderen behandelt. Mehrere Implementationsbeispiele auf den Gebieten der Fehlerbehandlung, der Datenverwaltung, des Kommandosprachinterpreters, des globalen Software-Entwurfs etc., werden vorgestellt, um die vielseitigen Einsatzmöglichkeiten von Ada in CACSD zu illustrieren.

Wegen der vielschichtigen Probleme auf dem Gebiet der CACSD werden in dieser Dissertation Themen von einem sehr breiten Spektrum der Ingenieurwissenschaften behandelt, insbesondere aber Themen der Regelungstechnik, des Software-Engineering, der formalen Sprachtheorie, der Rechnergraphik wie auch andere Aspekte der Informatik. Da die meisten Leser dieser Arbeit nur in einem paar dieser Gebiete spezialisiert sind, wird in jedem Abschnitt eine einführende Uebersicht gegeben, mit ausreichenden Hinweisen auf weitere Literaturstellen. Es werden jedoch einige Basiskenntnisse der Regelungstechnik und des strukturierten Programmierens vorausgesetzt.

Chapter 1

INTRODUCTION

Over the past 30 years, computers have gained, and they still continue to gain, an ever increasing importance in automatic control. Thereby, computer programs for automatic control have traditionally been constructed for two different purposes – to aid the control engineer during the *design* of a control system for a plant (the topic of this thesis), and to implement a real-time program for the digital *realization* of a control system. Correspondingly, the terms *off-line* and *on-line* programs are frequently used to classify control-related software.

Unfortunately, the terms on- and of-line are not mutually exclusive. For example, a real-time (on-line) data acquisition and/or identification package may be used during an otherwise off-line design cycle. Conversely, the automatic tuner of an adaptive controller may be seen as a design component build into an on-line program. To avoid this confused terminology, more distinctive terms have to be adopted.

Computer-Aided Control System Design (CACSD), a term which gained wide-spread usage only a few years ago (*Herget and Laub, 1982 and 1984*), was originally used to describe only the off-line *design* of controllers (*Cuenod, 1979 — Foreword by M. Mansour*). However, the actual analytical or numerical design of a controller is but one of many steps between the conception and the implementation of a control system, as illustrated by the complete control cycle in Figure 1.1 (adapted from *Mansour et al., 1985*).

In order to clarify the terminology used in this thesis, and thereby also to somewhat generalize the term CACSD according to more recent usage (*Jamshidi and Herget, 1985 — Foreword*), we define **CACSD** to be

> *the use of digital computers as primary tool during the modeling, identification, analysis and design phases of control engineering.*

Moreover, and again consistent with recent usage of the term, we define **CACSD packages** as being

> *stand-alone, interactive software programs to be used for CACSD.*

As depicted in Figure 1.1, these definitions are broad enough to encompass software for all phases of the design cycle except for the actual real-time implementation of the controller.

Alternative terms used by some authors to categorize approximately the same field as CACSD include Computer-Aided Engineering (CAE) (*Åström, 1984; Walker et al., 1984*) and Computer-Aided Control Engineering (CACE) (*Taylor et al., 1984*). These terms will, however, not be further used in this thesis.

Over the last half dozen years, several conferences have been held in the field (*Cuenod, 1979; Leininger, 1982; Hansen and Larsen, 1985; IEEE 1983, 1985 and 1986*). Also, some special collections of CACSD articles have been published in recent years (*Herget and Laub, 1982, 1984; CASCADE, 1984; Jamshidi and Herget, 1985*). Early projects in CACSD and related areas not covered by these publications have been documented by Wieslander (Man-machine interfaces, *1979*), Cellier (Modelling and simulation, *1979*) and Elmqvist (Modelling, *1978*).

The thesis will deal with man-machine communication interfaces as well as implementational issues of CACSD software packages. After a general excursion into the world of CACSD and its surroundings in Chapter 2, we will turn to the design of

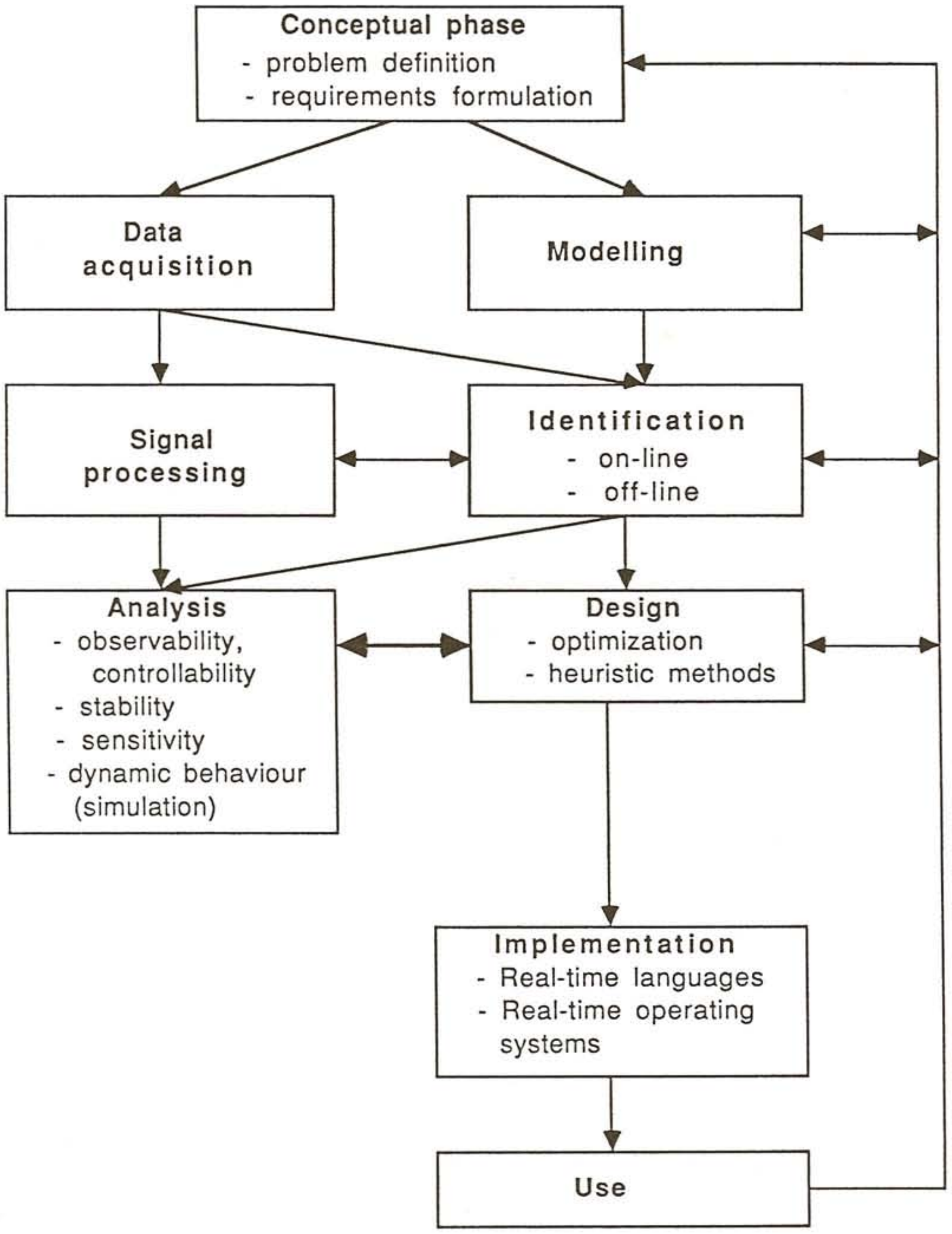


Figure 1.1: The design cycle in control theory.

flexible user-interfaces and the closely related aspect of including support for adequate data structures in Chapters 3 and 4. The increasing importance of software engineering concepts for the design and construction of large software packages is reflected in a detailed study of the suitability of Ada [1] (*ANSI, 1983*) as an implementation language for CACSD software in Chapter 5. Chapter 6 will conclude the thesis with a discussion of expected future trends in CACSD.

The design of modern CACSD packages is resolved in three main areas, namely control algorithms, man-machine interfaces and implementational deliberations. Each of these areas is imperative for a successful design. Moreover, the advances in each of the areas is dependent upon developments in other, related fields, as illustrated in Figure 1.2. Thus, aspects from the man-machine interface and implementational side of CACSD software design will be discussed in this thesis, and will be put in relation to their neighbouring non-control fields. It will be shown that these two areas are closely interrelated, and it will be argued that the hitherto frequently perceived incongruity between the interests of CACSD software *implementors* and CACSD package *end users* has essentially been obliterated with the advent of open, command-driven and user extendable CACSD tools. As a basis for this discussion, a flexible command language adapted to the needs of control engineers will be presented, and a comprehensive discussion of necessary data structures will be given.

The lack of a chapter on CACSD algorithms may in no way be seen as a depreciation of the importance of this field. Rather, the necessity of ever better control algorithms is reflected by the sections on the design of flexible, "plug-in" algorithmic interfaces of CACSD packages. However, the major thrust of this thesis is to create an environment which optimally supports the development and implementation of new control algorithms and strategies rather than the development of such algorithms by themselves.

Any pure paper-and-pencil research in the field of CACSD package design is bound to produce theoretically interesting, but impractical and/or unimplementable results. Therefore, this

[1]Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

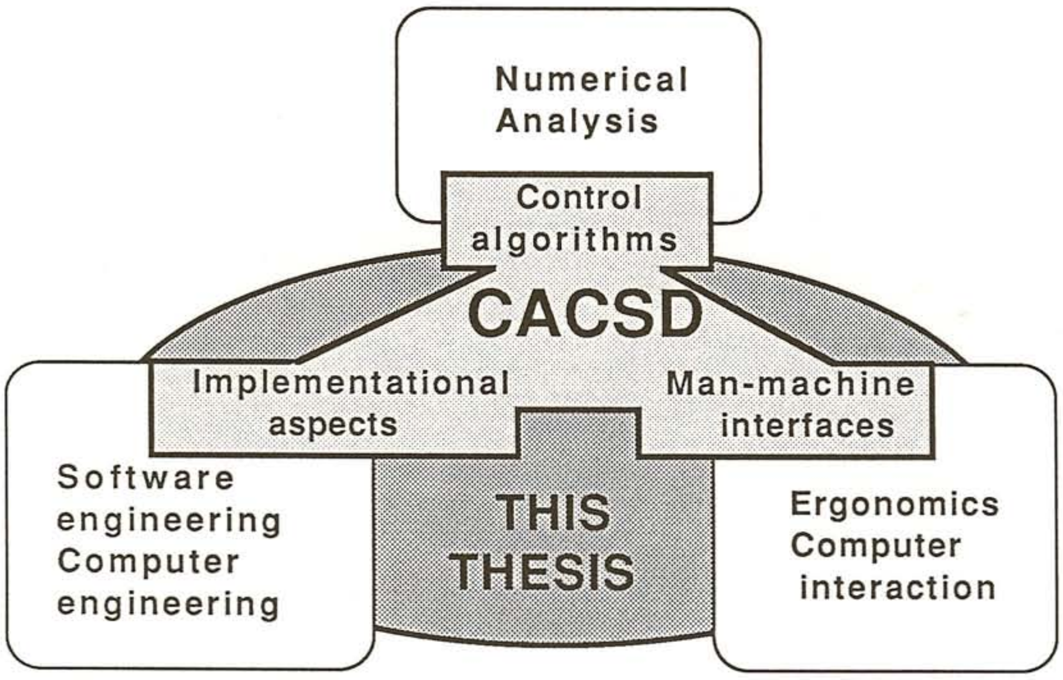


Figure 1.2: CACSD software components and related areas. The fields treated in this thesis are within the shaded area

work has been accompanied by the implementation of an advanced CACSD-package, IMPACT. All examples presented in this thesis, except those marked otherwise, are coded in the syntax of the new package IMPACT. It would be an exaggeration to call the present version of IMPACT operational though. The package has been developed as a research tool acting as a test-bed for the results of this thesis. However, it is not the framework of IMPACT that makes the software inoperable. The framework has been carefully designed, implemented, and tested. It is merely the lack of appropriate control algorithms currently implemented in IMPACT that makes the software still inoperable. Through plug-in incorporation of an adequate number of control algorithms, the package should become a well functioning CACSD product in the near future.

Chapter 2

GENERAL ASPECTS OF CACSD

2.1 CACSD in a global perspective

During the past decades, computational tools available to and design methods used by control engineers have wholly changed in pace with the rapid advances of digital computer technology. In particular, the graph-based frequency-domain methods used prior to the "computer-era" have been complemented with, and for some time overshadowed by, numerical, matrix-based state-space design methods. Lately, computers have given the traditional frequency-domain methods a renaissance through the use of new, computer-based graphical algorithms for multivariable controller design (*Frederick et al., 1985; Leininger, 1982*). This evolution of computational control-tools has been primarily spurred by developments in the following five fields:

- A tremendous development in *computer hardware*. During the past three decades, the processing power of digital computers (both with respect to their execution speed as well as their memory availability) have doubled every second to third year, whereas during the same time, the size as well as the price (per computational unit) of this hardware has been halved at approximately the same rate. This development has made several computational methods popular among control engineers that were previously known in theory, but were not practically usable, such as the digital simulation of continuous-time processes by means of numerical integration, as well as the iterative, on-line identification of physical processes. Also, new control methodologies have been initiated most of which are based on intricate numerical computations that were not feasible without computers (e.g. matrix-based methods such as Kalman filtering and state-feedback design).

- A slower, but equally impressive, development of *system software* such as operating systems, programming languages and program execution environments. These developments have been accelerated in recent years primarily due to the increasing cost of software as compared to hardware. In modern computer-systems, software cost amounts to 80% or more of the total installation cost (*Sommerville, 1985*)). Hence, results from newly developed fields such as software engineering must be employed to lower the overall system cost by increasing software productivity, ensuring software reliability, and assisting software maintenance.

- Better hardware and software for *graphical displays*. This development has already led to a renaissance of graphical control methods (e.g. the inverse Nyquist method). Moreover, utilization of fast bit-mapped displays will soon lead to a general acceptance of more flexible means of man-machine communication involving techniques such as pull-down menus, windowing, and graphical animation.

- *Quality numerical software.* Standard numerical routines which are efficient, accurate and reliable have been collected into wide-spread libraries for general use. Many of these routines were developed during the 70's and are now in the public domain (for example LINPACK (*Smith et al., 1974; Dongarra et al., 1979*), EISPACK (*Garbow et al., 1977*), MINPACK (*Moré et al., 1980*), ODEPACK (*Hindmarsh, 1983*), and ELLPACK (*Rice et al., 1985*)).

- New concepts as well as ready-to-use software in *related application fields* such as simulation and signal-processing have influenced and/or preempted developments in the field of CACSD.

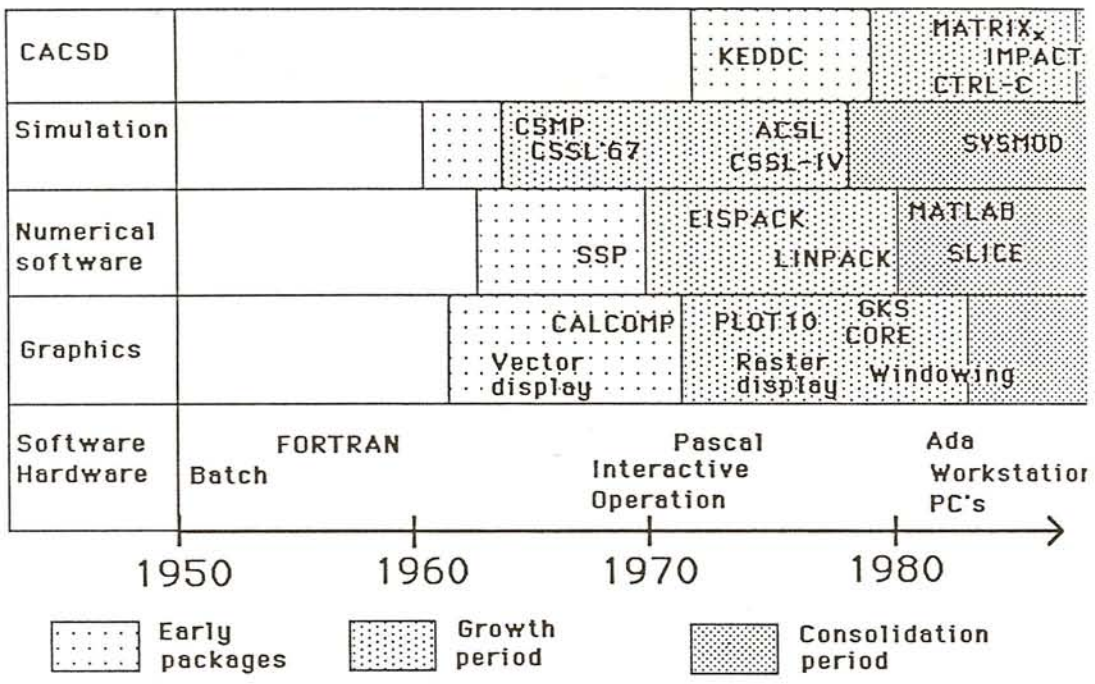


Figure 2.1: The "historical" development of *interactive* CACSD tools versus the availability of related (not necessarily interactive) software. A few products illustrate the prevailing state-of-the-art.

The relationship between CACSD software and these five fields is illustrated in Figure 2.1.

From this figure, the timely transfer of concepts from other fields to the control area can be easily envisaged. However, most of these related fields have been developed somewhat earlier. They are generally more mature, and have advanced beyond their phases of strongest growth into phases of consolidation. Therefore up to this point, the flow of ideas has been mostly unidirectional. For instance, the design of continuous simulation languages has been fairly well coordinated since the adoption of the CSSL'67 standard (*Augustin et al., 1967*). Contrary to this, the control community has only recently begun to discuss the need for world-wide guidelines in CACSD, and a first IFAC working-group for the development of CACSD-standards has just been formed (*IFAC, 1986*). Furthermore, recent developments in computer hardware, computer languages, and operating software should have wide-spread IMPACT (sic) on CACSD tools in the near future.

Despite the time-lag of CACSD in comparison with the other fields in Figure 2.1, fundamental results from the field of CACSD have already diffused into other disciplines. For example, many algorithms that originated in control are widely applicable also in other areas (Kalman-filters, Riccati-equation solvers, etc.). Also, some control-originated packages such as CTRL-C (*Little et al., 1984; CTRL-C, 1986*) and MATRIX$_X$ (*Shah et al., 1985*) find use in other application fields such as signal processing and simulation.

As CACSD matures, increased cross-fertilization between different application fields can be expected. Moreover, despite the relatively small size of the CACSD community, special purpose CACSD-systems based on customized hardware and system software may become feasible in the intermediate future (*Schmid, 1985*), just as special purpose simulation workstations and artificial intelligence workstations are currently emerging.

2.2 Computers in control systems design

As illustrated by Figure 1.1, the actual controller design is only one phase out of several in the total control cycle. Each of these phases require special algorithms operating on a wide range of different data structures.

- The first part of the iterative control cycle, the *conceptual phase* of problem formulation and requirements specification, is seldom supported by todays CACSD-packages. The major reason for this current lack in software support stems from the fact that decisions made during this project phase are mostly based on non-numerical operations and reasoning. The currently available CACSD packages lack both the data structures and the algorithms needed for symbolic processing. Recently, a tighter cooperation between control experts and expert system designers has been noticed. This link may soon produce software able to support the design engineer during this initial project phase.

- The analytical first phase of the control cycle, the *modeling* phase, is the least uniform among the phases, and thereby it is also the step that can least easily be automated. The user may deal with linear as well as nonlinear systems, continuous as well as discrete/sampled-data systems, systems with known as well as unknown parameters, the models may be self-containing, or they may be hierarchically organized, and so on. Due to the abundance of different model types, the user interface must support a large variety of model-representation structures as well as algorithms for general model interconnections. Hitherto, the most versatile modeling capabilities have been found in general purpose simulation languages such as ACSL (*1986*) or SYSMOD (*Baker et al., 1983; SYSMOD, 1986*).

- The *data acquisition* was traditionally performed over an analog data-logger and transferred to the computer in a manual or semi-automatic fashion over e.g. a digitizing tablet. Today, stand-alone data-logging hardware or special purpose, combined software–hardware products for process computers or even small PC's will perform this task automatically. These systems may include *signal processing* elements for data compression and signal analysis, or the raw data is passed on to some off-line signal processing program. The real time acquisition of data is not explicitly treated in this thesis, however, the data structures presented in Section 4.7 cover the requirements of the data acquisition and signal processing phases as well.

- During the *parameter identification* phase, the parameters of a given model structure are estimated from measured data by the use of some optimization techniques. In addition to a parameterized model representation, the identification algorithms require signal representation(s) as input.

- The central part of any CACSD package contains the algorithms for model *analysis* and controller *design*. Depending on the structure of the model and the design methodology used, the algorithms used and the structures operated upon may vary greatly. Most CACSD packages work on linear system representations, and design tools operating in the time (state-space) and/or the frequency domain. Correspondingly, data structures to represent ordinary (eventually complex) matrices as well as rational-function matrices must be supported. Nonlinear simulation, model linearization, model reduction, and model transformations are other operations needed during this project phase.

- The *implementation* phase does not, according to our definition, belong to the sphere of CACSD. Nevertheless, different attempts to add automatic generation of real-time code to traditional CACSD tools (e.g. *Lehman et al., 1986*) have been made lately. However, it was never tried in any of these attempts to incorporate real-time software *within* large, integrated CACSD tools. This field is certainly worth a closer study, and, although not explicitly treated in this thesis, the parallel capabilities described in Chapter 5 together with the data-structures described in

Chapter 4 could form a solid base for any further work in this direction.

Despite the large range of data representations and algorithms needed during the different phases of the control-cycle, there are some common traits: data structures in CACSD software usually represent systems or signals in some form, and/or they are used to store intermediate results in matrix/symbolic form. Hence, CACSD packages can be seen as a blend of

- *numerical software* for the implementation of control algorithms. Many of these algorithms, and particularly those working on systems described in the time-domain, contain standard linear-algebra routines as their basic building blocks. Other control algorithms are based on polynomial operations or numerical integration. The trend is to rely on standard numerical software for economy and reliability.

- *symbolic software* for defining and manipulating nonlinear systems. Traditionally, symbolic software was either hidden in simulation languages for the translation of a parallel model description (ODE's) to a sequential and thereby executable program, or directly used in general symbolic packages for algebraic equation manipulations (*Wolfram, 1985*). In some newer control packages, linearization algorithms are used to connect (symbolic) modeling languages to (numeric) design/analysis parts.

  None of the presently wide-spread CACSD tool give direct access to pure symbolic manipulations, and few implementations based on numeric software can be easily extended to facilitate symbolic descriptions. Nevertheless, the use of symbolic manipulations in CACSD will probably increase in the future.

- *"intelligent" software* for the use of expert systems within control packages. The term "expert systems" has undoubtedly become the newest buzzword in all computer fields. Nevertheless, the current expert system development must be taken seriously. Pioneer work has shown that expert systems can be meaningfully employed in a control environment (*James et al., 1985;, Taylor and Frederick, 1984*).

- *graphical software.* Practically all modern CACSD packages support graphical display of time-histories (simulation) and/or frequency plots (design in the frequency domain etc.). Moreover, a few packages allow the user to enter system descriptions graphically in a block diagram form (*King et al., 1984; Elmqvist et al., 1986*). Neither of these applications call for the highest-resolution graphical displays (as they are used e.g. in VLSI-design) or for the highest-speed communication links, yet the advent of modern workstations with fast and high-quality graphics opens a wide range of new possibilities. For instance, it will be feasible to support multi-windowing, "on-line" animation of control processes, and interactive "on-line" tuning of controller parameters. This has been illustrated by the experimental teach-ware program shown in Figure 2.2 (*Schaufelberger et al., 1986*).

  Graphical output does not suffer from the limitations of alphanumerical terminals in displaying indices, etc. We can therefore expect to get better readable output of for instance polynomials and system descriptions on modern workstations.

- *parser software and user interfaces.* Most early developers of CACSD software concentrated their efforts on numerical problems and thereby neglected the development of good (that is: flexible and robust) man-machine communication interfaces. In recent years, the design of powerful user interfaces has become the dominant issue in the development and/or assessment of CACSD tools (*Bongulielmi and Cellier, 1984; Jamshidi and Herget, 1985*).

While early control packages contained up to 80% numerical software (*Agathoklis, 1979, 1986*), modern CACSD tools display a more balanced distribution between the different software components. Figure 2.3 shows the estimated relative size of the different software components in IMPACT. Other modern packages exhibit similar distributions.

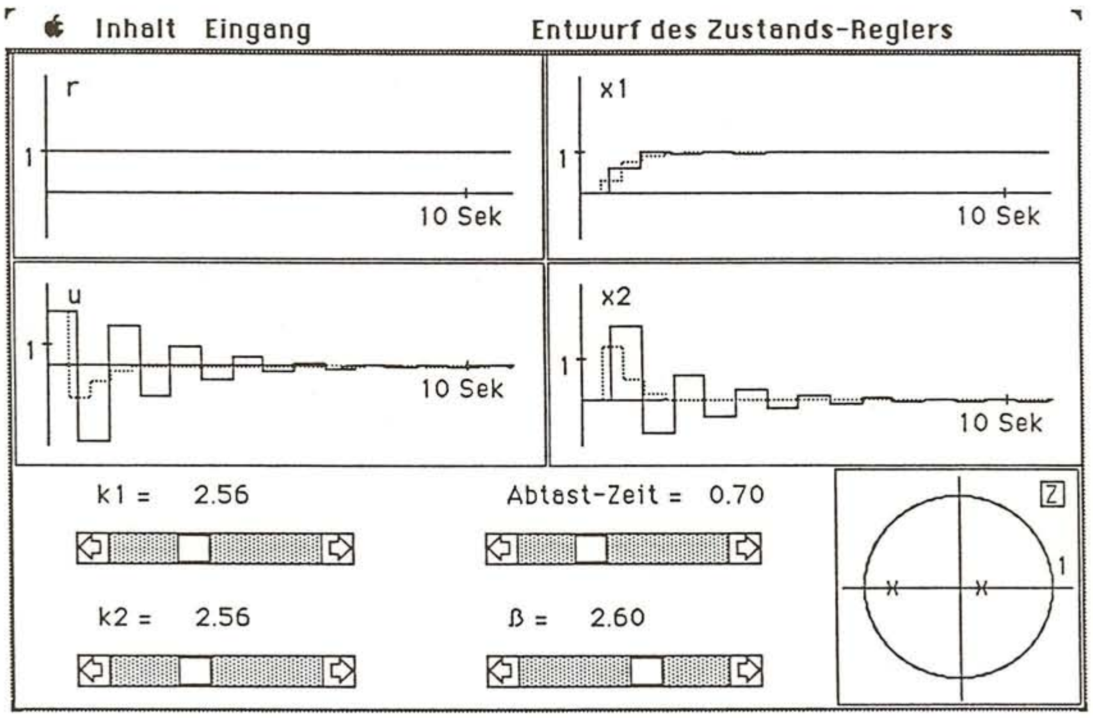


Figure 2.2: Mouse controlled program illustrating state-feedback controller design. This experimental program is used in classroom exercises and runs on a MacIntosh.

Obviously, multifaceted knowledge is needed to design and develop a modern CACSD package. The package developer can partly rely on the specific knowledge of others (by using precoded commercial or public domain software). This is, however, only possible when the imported software can be used as a "black box". In particular, the development of a suitable user-interface will remain a major software undertaking until suitable kernel systems such as IMPACT or the system described by Goodfellow and Munro (*1985*) become widely available. Such a major development must involve people with knowledge in all of the above mentioned fields and, most important of all, experience in software engineering.

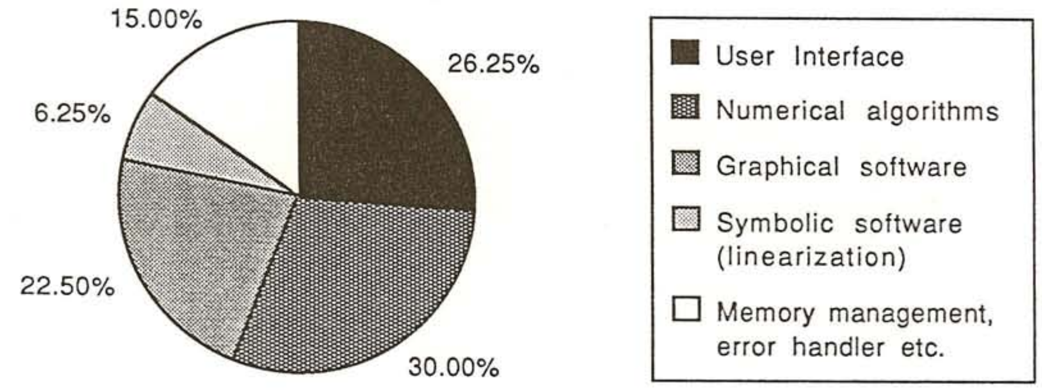


Figure 2.3: Relative size of different code segments of IMPACT

2.3 Integrated CACSD-facilities

We have seen that CACSD tools exist for all phases of the design sequence, but that each single package normally is suited for only one or two of these phases. However, particularly in a student environment, the use of a single, multi-purpose package is advantageous. Conceptually, the development of such a multi-purpose CACSD facility can be approached on three different levels:

- several independent programs communicating through a common data base
- several control packages connected via a common data base and a common user interface
- a fully integrated CACSD package

The simplest approach is the one using a common data base through which intermediate data can be transported from one part of the facility to another. It is mainly used when already existing packages are to be interconnected (a typical example is the link between the CTRL-C design package and the ACSL simulation language (*CTRL-C, 1986*), another example is the Federated Computer-Aided Control Design System from General Electric, which combines four different CACSD packages (*Spang, 1985*)). Major drawbacks of this approach are that the user has to get acquainted with several different systems which are conceptually different and which do not provide for consistent data structures, and that information is often lost in the transfer or has to be added after the transfer (a typical example is the run-time information which is needed by the analysis and nonlinear simulation modules of the CACSD software, but which is redundant in and incompatible with the linear design module).

The approach of combining many individual programs over a common user interface has been successfully used in several larger CACSD projects, for example in the Lund control suite surveyed by Wieslander (*1980*) and Åström (*1983*). This suite is built upon a common macro handler (INTRAC). The macro handler gives the user access to a basic set of general commands (edit/execute macros, change/show parameters). The other parts of the package add specific context-adapted commands, like commands for identification (IDPAC), polynomial operations (POLPAC) and simulation (SIMNON). Using this approach, the user needs to know only one system - the common interface. A further advantage is that the individual parts of the package can be designed quite independently of each other using their own data structures. However, the context-dependency of the available commands and data structures makes an overview of the full package and all its possibilities difficult, especially for the beginner.

The fully integrated approach differs from the previous ones in that the user is presented with a single, context-independent interface from which all phases of the design sequence can be performed. This means that, at any moment, the user is presented with a much more powerful, but not necessarily more intricate, tool. In fact, the very integration of the different segments into one interface can make the system easier to use. For example, it is possible to design the package in such a way that the commands for the simulation of linear and nonlinear systems look exactly the same, despite the fact that these simulations have nothing in common from an algorithmic point of view.

In our opinion, this third approach is to be preferred. Not only does the integrated interface shorten the total familiarization time, it also precludes all errors deriving from conceptual differences in the packages (e.g. varying specifications of closed/open loops or different tolerance indications). An example of such an integrated system is IMPACT.

IMPACT will give access to algorithms for all parts of the design sequence (except for the data acquisition and real-time implementation phases) using a common set of data structures. Moreover, in addition to the standardized interface to all algorithms, IMPACT provides the user with a full-fledged algorithmic command language and a wide selection of applicable data structures. This makes the package user extendable and therefore usable not only in education but also in a research environment.

2.4 The educational value of CACSD

Design is by its very nature an iterative process. Several trials have to be made until a reasonable control system results. If done by pencil and paper, this requires much work (drawing frequency responses, calculating dominant pole responses, etc.), and the methods used are only approximate ones. This can be considerably improved by the use of computers. Methods that are not important for the understanding of the subject matter (e.g. the manual drawing of Root-Locus curves) can be removed, giving time for the introduction of additional control concepts. Thereby, the essential facets of control engineering can be brought to bear over a shorter time-span.

Moreover, many problems arising in control theory are solvable only by numerical methods. Through the use of CACSD packages, these problems can now, for the first time, be treated in the exercises of intermediate control courses. For example in conventional paper-and-pencil exercises, the students almost exclusively work on linear (linearized) systems. Through the use of CACSD packages supporting linear design techniques as well as nonlinear simulations, the students can compare the system behaviour of the regulated linearized system with that of the regulated original, nonlinear process after the controller design.

However, although the use of CACSD indisputably opens up new perspectives in control education, an indiscriminate replacement of conventional exercises with CACSD sessions may prove contra-productive. Several manual schemes, which can be performed much faster using numerically better suited methods on a computer than by hand, often help the students to get a better understanding of the underlying theories during their use. For example, many CACSD packages support algorithms transforming systems from the time to the frequency domain and vice versa. Nevertheless, we feel that it is pedagogically better to let the students perform this translation by hand a few times using simple examples, as this provides a better feel for the physical meaning of the translation. Moreover, the student should always have means to roughly verify the results he gets from the computer.

From our own experience, we conclude that the use of CACSD tools enables an enhancement of the overall scope of a course, but that during carelessly formulated CACSD exercises, the students often overlook the actual theoretical/numerical difficulties and loose the overview of the treated subject matter.

Chapter 3

MAN-MACHINE INTERFACES IN CACSD

3.1 Introduction

In Section 2.1, we discussed how the rapid advances in several areas of computer hardware and software have influenced the development in control theory, and how this has triggered a dramatic expansion of methods and tools available to control engineers. However, from the perspective of a control engineer using the computer only as a tool, the impact in these five areas has neither been uniform nor simultaneous. A control engineer assigned to solve a particular problem using a given computer tool will first and foremost be confronted with the user interface of the tool (independent of whether this tool is an interactive package, a library of algorithms, or simply a compiler/linker). Only when this user-interface has been mastered will other problems (like computing speed, dimensional limitations or insufficient accuracy) appear.

Obviously, an easy-to-use and flexible user-interface is a most important prerequisite of any (control-oriented) computer-tool to be used efficiently, yet the development of such interfaces has for a long time lagged behind the advances in other areas such as robust methods and efficient algorithms. Let us illustrate this with an example:

Assume a control engineer wishes to calculate the sets of eigenvalues and the inverse modal matrices (the inverse of matrices with the eigenvectors as their columns) of several 5*5 system matrices.

Forty years ago, he would have needed a lot of paper and even more patience to solve this problem (on the other hand at this time, state-space methods were not used, so no control engineer would have felt the urge to solve such a problem).

Twenty years ago, our control engineer might have had a digital computer at his disposal. However, he would have had to write by himself a program which calculated the eigenvectors and inverted the modal matrices.

Ten years ago, most control engineers had access to some libraries containing standard mathematical algorithms, such as SSP (*IBM, 1968*), LINPACK (*Dongarra et al., 1979*), EISPACK (*Smith et al., 1974; Garbow et al., 1977*), and IMSL (*1982*). Nevertheless, our engineer still had to construct a program to read the matrices, call the algorithm-routine(s) and print out the results. Figure 3.1 lists a minimal program solving this problem through calls to IMSL routines, together with a sample input and output. A lot of time was lost before the user knew which library routines to call, what values to give the numerous parameters of these routines and how to compile and link this program with the correct library routines. More time was then lost until the input format corresponded to the input data, the result was properly displayed and all small programming errors had been detected. At the end of this tedious work, the engineer had a re-usable, but inflexible, special-purpose program.

Five years ago, the situation was again slightly improved. Several *interactive* numerical control packages had become available. These allowed the user to enter his data and perform a limited number of operations without writing any programs of his own. Unfortunately, the form employed for the user-dialogue was seldom very efficient. More often than not, a rigid question-and-answer dialogue had to be used. Such a conversation is illustrated in Figure 3.2 by an example from INTOPS, a package developed by Grepper and coworkers (*1977*) for use in undergraduate control exercises at the ETH. This tedious kind of conversation slowed down the use of these packages and gave the user very little flexibility in the actions he could take.

```
C*****PROGRAM TO 1/ READ A SYSTEM MATRIX
C*****           2/ CALCULATE THE MODAL MATRIX
C*****           3/ CALCULATE THE INVERSE MODAL MATRIX
      COMPLEX CMAT(100),CVAL(10),CVEC(100)
      REAL RVEC(100),RINV(100)
      CALL READA(N,CMAT)
      CALL MODAL(N,CMAT,CVAL,CVEC)
      CALL WRITM(N,CMAT,CVAL,CVEC)
      DO 10 I=1,N*N
        RVEC(I)=REAL(CVEC(I))
        IF(AIMAG(CVEC(I)).NE.0.0)GOTO 11
   10 CONTINUE
      CALL INVMO(N,RVEC,RINV)
      CALL WRITI(N,RINV)
      STOP
C
   11 WRITE(6,100)
  100 FORMAT(40H IMAGINARY EIGENVECTORS, CAN NOT PROCEED)
      STOP
      END
C
      SUBROUTINE READA(N,CMAT)
C*****READ A REAL SQUARE MATRIX ROWWISE
      COMPLEX CMAT(1)
      REAL CINP(10)
      READ(5,100)N
      IF(N.GT.10) GOTO 11
      DO 10 I = 1,N
      READ(5,101)(CINP(J),J=1,N)
      DO 5 J = 1,N
      CMAT((J-1)*N+I) = CMPLX(CINP(J),0.0)
    5 CONTINUE
   10 CONTINUE
      RETURN
C
   11 WRITE(6,102)
      STOP
  100 FORMAT(I5)
  101 FORMAT(10F10.0)
  102 FORMAT(33H DIMENSIONAL LIMIT OF 10 EXCEEDED)
      END
```

(Continued)

```
      SUBROUTINE MODAL(N,CMAT,CVAL,CVEC)
C*****CALCULATE THE MODAL MATRIX
      REAL WK(250)
      CALL EIGCC(CMAT,N,N,2,CVAL,CVEC,N,WK,IER)
      IF(IER.NE.0)GOTO 11
      RETURN
C
   11 CONTINUE
      WRITE(6,100)
  100 FORMAT(35H ERROR CALCULATING THE MODAL MATRIX)
      STOP
      END
C
      SUBROUTINE INVMO(N,RVAL,RINV)
C*****CALCULATE THE INVERSE OF A MATRIX
      REAL RVAL(1),RINV(1),WKAREA(250)
      IDGT = 0
      CALL LINV2F(RVAL,N,N,RINV,IDGT,WKAREA,IER)
      IF(IER.NE.0)GOTO 11
      RETURN
C
   11 CONTINUE
      WRITE(6,100)
  100 FORMAT(33H ERROR INVERTING THE MODAL MATRIX)
      STOP
      END
C
      SUBROUTINE WRITM(N,CMAT,CVAL,CVEC)
C*****PRINTS SYSTEM MATRIX, EIGENVALUES AND MODAL MATRIX
      COMPLEX CMAT(1),CVAL(1),CVEC(1)
      CALL USWCM(14H SYSTEM MATRIX,14,CMAT,N,N,N,2)
      CALL USWCV(12H EIGENVALUES,12,CVAL,N,1,2)
      CALL USWCM(13H MODAL MATRIX,13,CVEC,N,N,N,2)
      RETURN
      END
C
      SUBROUTINE WRITI(N,RINV)
C*****PRINTS THE INVERSE MATRIX
      REAL RINV(1)
      CALL USWFM(17H INVERSE MODAL M.,17,RINV,N,N,N,2)
      RETURN
      END
```

(Continued)

(Input to the program)

```
  5
       18          0          0         10        -10
       20         -7         -3          7        -13
       10         -3         -3          2         -8
      -10          0          0         -6          5
       10          0          0          5         -5
```

(Partial output from the program)

```
 EIGENVALUES

    (       5.3375,        0.0000)  (       -8.6056,        0.0000)
    (       2.4326,        0.0000)  (       -0.7702,        0.0000)
    (      -1.3944,        0.0000)
 MODAL MATRIX
                    1                              2
                    3                              4
                    5

  1 (       0.5000,        0.0000)  (        0.0000,        0.0000)
    (       3.1856,        0.0000)  (       -0.1381,        0.0000)
    (       0.0000,        0.0000)

  2 (       0.2580,        0.0000)  (        8.0000,        0.0000)
    (       1.1685,        0.0000)  (        0.2013,        0.0000)
    (      -3.3282,        0.0000)

  3 (       0.1077,        0.0000)  (        4.2815,        0.0000)
    (       0.1875,        0.0000)  (       -0.9543,        0.0000)
    (       6.2188,        0.0000)

  4 (      -0.2894,        0.0000)  (        0.0000,        0.0000)
    (      -2.0568,        0.0000)  (        0.3695,        0.0000)
    (       0.0000,        0.0000)

  5 (       0.3437,        0.0000)  (        0.0000,        0.0000)
    (       2.9023,        0.0000)  (        0.1103,        0.0000)
    (       0.0000,        0.0000)
```

Figure 3.1: FORTRAN program calculating the eigenvalues of a system and the inverse of the corresponding modal matrix. The program uses the IMSL library. Note that IMSL does not contain a subroutine for calculating the inverse of a complex matrix (!).

Today, many control engineers have access to modern CACSD packages offering easy-to-use and yet flexible command-driven user interfaces as illustrated in Figure 3.3. Comparing this figure with the previous ones, it is easy to envision how the time needed for a simple eigenvalue computation has been reduced by a factor of 100 (10) during the last ten (five) years.

The described evolution is true not only for innovative developments at exclusive sites, but also for the broad, worldwide development of CACSD software. However, there is a 5-6 year delay until the mainstream of CACSD developments catch up with the forefront. This delay becomes obvious from studying overviews such as ELCS, a regularly published Newsletter which lists and describes control software packages (*ELCS, 1986*). Therefore, many of the concepts described in this thesis cannot be expected to be in *common* use for some years to come (cf. Section 6.2 on standardization).

This chapter will deal with properties of easy-to-use, efficient and yet flexible user interfaces to software implementing control algorithms. Thereby, we will start by differentiating between interactive control packages and batch-oriented control programs. Thereafter, we will present different approaches to interactive interfaces. The bulk of the chapter will discuss the individual elements of such an interface.

3.2 Batch versus interactive CACSD software

In ELCS (*ELCS, 1986*) and elsewhere, a clear distinction is made between interactive CACSD packages and control-oriented subroutine libraries. The latter are source-/object-code libraries intended to be included in user written programs, which then are executed in a batch fashion. This thesis will deal with CACSD packages which can be run *interactively* according to the following, CACSD-relevant definitions:

- Batch-oriented *computer programs* are defined as programs which execute autonomously (possibly using a predefined

```
INTOPS>

I>      FOR SELECTING PROGRAM OPTIONS:

        OPTION = POLOPS (POLINOMIAL OPERATIONS), OR
        OPTION = MATOPS (MATRIX OPERATIONS), OR
        OPTION = LTDOPS (LINEAR TIME DOMAIN OPERATIONS),

I>      OPTION = MATOPS

MATOPS>

M>      OP CODE = ENTER
M>      NAME = A
M>      COMMENT = A-MATRIX
M>      ROWS = 5
M>      COLUMNS = 5

M>      A    ( 1, 1) = 18
M>      A    ( 1, 2) = 0
M>      A    ( 1, 3) = 0
M>      A    ( 1, 4) = 10
M>      A    ( 1, 5) = -10
M>      A    ( 2, 1) = 20
M>      A    ( 2, 2) = -7
M>      A    ( 2, 3) = -3
M>      A    ( 2, 4) = 7
M>      A    ( 2, 5) = -13
M>      A    ( 3, 1) = 10
M>      A    ( 3, 2) = -3
M>      A    ( 3, 3) = -3
M>      A    ( 3, 4) = 2
M>      A    ( 3, 5) = -8
M>      A    ( 4, 1) = -10
M>      A    ( 4, 2) = 0
M>      A    ( 4, 3) = 0
M>      A    ( 4, 4) = -6
M>      A    ( 4, 5) = 5
M>      A    ( 5, 1) = 10
M>      A    ( 5, 2) = 0
M>      A    ( 5, 3) = 0
M>      A    ( 5, 4) = 5
M>      A    ( 5, 5) = -5
```

(Continued)

```
 INPUT MATRIX    A         5 ROWS     5 COLUMNS

 18.000      0.00000     0.00000       10.000      -10.000
 20.000     -7.0000     -3.0000        7.0000     -13.000
 10.000     -3.0000     -3.0000        2.0000      -8.0000
-10.000      0.00000     0.00000      -6.0000       5.0000
 10.000      0.00000     0.00000       5.0000      -5.0000

M>      OP CODE = EIGVAL
M>      NAME = A


M>      THE EIGENVALUES AND -VECTORS ARE REAL

 EIGENVALUES               5 ROWS     1 COLUMNS

  5.3375
  2.4326
-0.77016
 -8.6055
 -1.3944

M>      DO YOU WANT TO SAVE THE EIGENVALUES (YES OR NO): Y
M>      REAL PARTS (YES OR NO): Y
M>      NAME = EVAL
M>      COMMENT = EIGENVALUES
M>      MATRIX SAVED
M>      IMAGINARY PARTS (YES OR NO): N
M>      DO YOU WANT TO SAVE THE EIGENVECTORS (YES OR NO): Y
M>      REAL PARTS (YES OR NO): Y
M>      NAME = EVEC
M>      COMMENT = EIGENVECTORS
M>      MATRIX SAVED
M>      IMAGINARY PARTS (YES OR NO): N


M>      OP CODE = LIST
M>      NAME  N * M  COMMENT

M>      A     5   5  A-MATRIX
M>      ECAL  5   1  EIGENVALUES
M>      EVEC  5   5  EIGENVECTORS
```

(Continued)

```
M>      OP CODE = SHOW
M>      NAME = EVEC

 SELECTED MATRIX EVEC     5 ROWS     5 COLUMNS

 0.43022       -2.4148    -0.15262   -0.10803E-06   0.33713E-06
 0.22199      -0.88577     0.22243    -10.984         3.3625
 0.92644E-01  -0.14210    -1.0546     -5.8788        -6.2829
-0.24905        1.5592     0.40834    0.23096E-06  -0.55506E-06
 0.29571       -2.2001     0.12186   -0.89407E-07   0.96857E-07

M>      OP CODE = MINV
M>      INPUT MATRIX TO BE INVERTED = EVEC
M>      RESULT IN WORKING MATRIX
M>      MATRIX OUTPUT (YES OR NO) : Y

 WORKING        MATRIX    5 ROWS     5 COLUMNS

 12.079       0.11507E-06 -0.36006E-07   6.9926       -8.3027
 1.6769       0.26701E-07 -0.14933E-07   1.0827       -1.5278
0.96416       0.98268E-07 -0.11169E-06   2.5795        0.76981
0.94668E-01 -0.70766E-01 -0.37873E-01 -0.17421E-02 -0.74206E-01
-0.11024      0.66215E-01 -0.12372      -0.35276      -0.14766

M>      OP CODE = SAVE
M>      NAME = INVE
M>      COMMENT = INVERSE MODAL MATRIX
M>      MATRIX SAVED

M>      OP CODE = LIST
M>      NAME   N * M  COMMENT

M>      A      5   5  A-MATRIX
M>      EVAL   5   1  EIGENVALUES
M>      EVEC   5   5  EIGENVECTORS
M>      INVE   5   5  INVERSE MODAL MATRIX
```

Figure 3.2: Question-and-answer dialogue of INTOPS for calculating the eigenvalues of a system and the inverse of the corresponding modal matrix. The dialogue has been slightly modified to fit onto three pages (!).

```
<>
a = <18  0  0 10 -10
     20 -7 -3  7 -13
     10 -3 -3  2  -8
    -10  0  0 -6   5
     10  0  0  5  -5>;
<>
help eig

EIG   Eigenvalues and eigenvectors.
      EIG(X)  is  a  vector  containing the eigenvalues of a
      square matrix  X .
      <V,D>  =  EIG(X)  produces  a  diagonal  matrix  D  of
      eigenvalues  and a full matrix V whose columns are the
      corresponding eigenvectors so that  X*V = V*D .
<>
<evec,eval> = eig(a)

EVAL  =
   -8.6056    0.0000    0.0000    0.0000    0.0000
    0.0000    5.3375    0.0000    0.0000    0.0000
    0.0000    0.0000    2.4326    0.0000    0.0000
    0.0000    0.0000    0.0000   -0.7702    0.0000
    0.0000    0.0000    0.0000    0.0000   -1.3944

EVEC  =
    0.0000    0.7432    5.1403    0.1532    0.0000
   -0.8817    0.3835    1.8855   -0.2232    1.4190
   -0.4719    0.1600    0.3025    1.0585   -2.6515
    0.0000   -0.4302   -3.3189   -0.4098    0.0000
    0.0000    0.5108    4.6832   -0.1223    0.0000
<>
invm = inv(evec)

INVM  =
    1.1795   -0.8817   -0.4719   -0.0217   -0.9245
    6.9924    0.0000    0.0000    4.0479   -4.8063
   -0.7878    0.0000    0.0000   -0.5087    0.7178
   -0.9607    0.0000    0.0000   -2.5703   -0.7671
   -0.2612    0.1569   -0.2932   -0.8359   -0.3499
```

Figure 3.3: Command-driven dialogue of MATLAB for calculating the eigenvalues of a system and the inverse of the corresponding modal matrix.

set of input-files), whereas interactive programs communicate with the user during the execution (e.g. to determine the next action to be taken).

- Correspondingly, *human operators* may start programs in a batch or interactive fashion. Batch programs will run autonomously, unless the user crudely interrupts the program by pressing CTRL-C, pulling the power cord, etc.

- For our purposes, it is enough to define the *operating system* of a computer to be interactive if interactive programs can be run on them, and batch-oriented otherwise.

Obviously, interactive programs can only run on computers having an interactive operating system. However, the inverse is not true. For example, the batch program in Figure 3.1 could have been developed on an interactive computer using an interactive editor, and yet it would execute in a batch fashion. Moreover, interactive programs such as the one in Figure 3.3 can be executed in a batch fashion by specifying that the input is to be read from an external file rather than from the terminal.

- Yet another, slightly misleading, use of the notations "interactive" and "batch" is based on the *program development.* Thereby, all programs where the user has to construct program code of his own (by hand-coding or through automatic code generation) to be compiled and linked/loaded (possibly together with extensive libraries) are called batch-programs.

This last classification is somewhat inaccurate, as thereby interactively controllable programs with user-coded components (e.g. an additional algorithm) still would be considered batch programs. We will therefore refrain from this classification. Instead, we will use the term *hard-coded algorithms/subprograms* to indicate program code implemented by the user and linked to an already existing package (as opposed to so-called *soft-coded subprograms*, to be defined later).

3.3 Modes of interaction

When designing a new interactive system, one of the first actions must be to decide in which form the man-machine interaction is to take place. This decision should not be taken lightheartedly, as the mode of interaction determines the user-friendliness, and thereby also user acceptance, of the system; although this interface is not the brain of any CAD-system, it certainly serves as both eyes and mouth.

Moreover, the chosen mode of interaction influences the structure of the kernel controlling the package. In particular, the data-structures of the kernel are very closely knitted to the user interface. As any late changes in the central data structures are the worst of all possible nightmares for any software developer, the design of the interactive interface should be done carefully, so that no later modifications need to be made.

Apart from some still exotic ways of communication, like speech input and natural language input, five basically different ways of interactive input exist:

- graphical input.
- question-and-answer method
- menu-driven operation
- form-driven input
- command-language communication

In the next sections, we will shortly discuss these five possibilities and their application to CACSD packages.

3.3.1 Graphical input

Graphical input is particularly interesting for specifying system descriptions (e.g. models of physical processes and controllers) in topological form. Figure 3.4 depicts a fictitious system for entering topological/equational system descriptions using a mouse

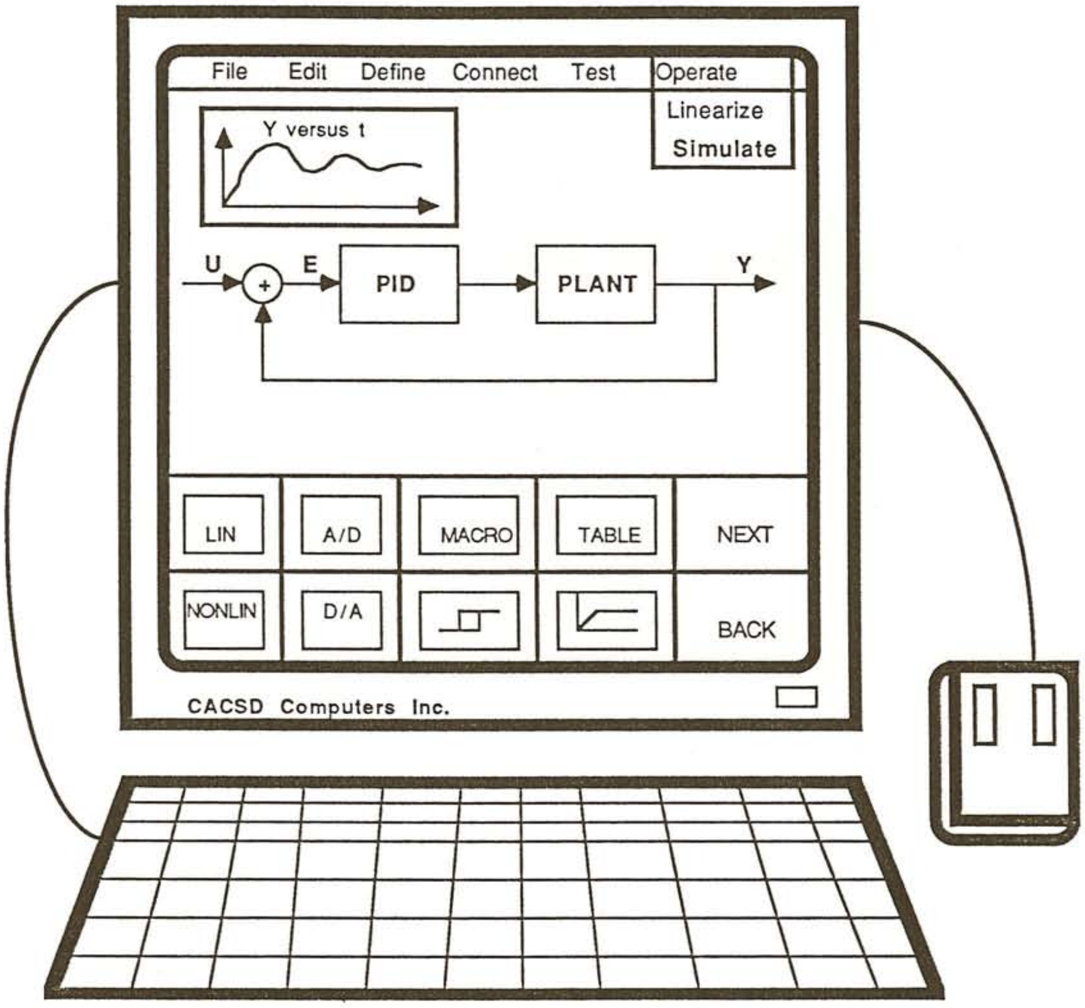


Figure 3.4: Fictious graphical user-interface for entering system topologies.

and a keyboard. Similar academic/commercial software packages exist for entering control systems (SYSTEM_BUILD – *Shah et al., 1985*; MODEL-C – announced companion to CTRL-C; SAICAD – *King and Gray, 1984*; DOSU – *Domeisen et al., 1985*) discrete network-simulation systems (TESS – *1986*; CINEMA – *1986*) and general, hierarchical topological systems (Hibliz – *Elmqvist et al., 1986*).

User acceptance of packages using a graphical input is quite varied; on the one hand, the replacement of a formal modeling language with a self-explanatory mouse-driven operation is much appreciated, but on the other hand, implementational limitations of some of these early packages restrict their areas of application. Especially for the modeling of large systems, the lack of hierarchical concepts in DOSU, SAICAD, and TESS leads to extremely large and messy graphs, whereas the limitation to seven elements per hierarchical level of PC-based versions of SYSTEM_BUILD limits the possible connections.

A more fundamental critique of graphical input is that it is well suitable for *describing and manipulating* topological systems, but that *operations using* these systems become quite cumbersome and slow if performed in a purely graphical manner. Hence, some of the here mentioned packages (e.g. TESS, CINEMA, and SYSTEM_BUILD), have automatic conversions of systems to representations accessible over alphanumeric interfaces. These alphanumeric, command-driven interfaces are then used for the "execution", that is simulation, analysis or synthesis, of the systems. This combination of graphical definition parts and command-driven execution parts form very powerful, integrated packages.

Although well-implemented graphical-input systems operating on modern workstations are very easy to use, they are sometimes criticized by skilled personnel for being considerably slower than a corresponding alphanumeric input when the description of a large system is to be entered. However, graphs still have an enormous documentation value. It would therefore make sense to allow for alphanumeric as well as graphical input for system definitions and to add yet another module to generate the graph out of the coded program. This would also allow the user to delete the graph from the data base to save memory.

The hitherto mentioned, presently available, graphical systems all have one serious drawback; they are highly hardware dependent. Most of these packages will run on only one particular kind of computer using a specific (sometimes hardware-extended) brand of terminal. This incompatibility with alternative hardware can only be avoided through consistent use of a graphical standard such as GKS (*ANSI, 1985*) and Core (*ACM/SIGGRAPH, 1979*).

3.3.2 Question-and-answer dialogue

The four interaction-modes described in the following are all operating on alphanumerical information. This means that they, at least in principle, can be implemented to run on a variety of computers and terminals in a fairly portable manner. However, "fancy" implementations of these conversational modes using for example graphical pull-down menus or direct cursor control may again be strongly hardware-dependent.

Question-and-answer interaction has a long tradition in interactive application-programming, including CACSD. A majority of the CACSD-packages developed during the seventies had a question-and-answer interface (*ELCS, 1985*) similar to the one illustrated in Figure 3.2 There were several reasons why this kind of interface was so widely spread:

- It is simple to program a question-and-answer dialogue, as the text to be displayed on the screen can be statically coded in the subroutine needing the information. Also, the interpretation of user input is straight forward, as this input has a simple structure and therefore does not need to be parsed and decoded.

- As the program keeps the initiative at all times, such a system is extremely simple to use. The user is rarely left with any complicated choices, instead, he simply needs to enter concise answers to simple questions in form of one number or one name.

This last "advantage" is at the same time the major disadvantage of question-and-answer dialogues: the dialogue is totally controlled by the computer, and no deviation from the pre-programmed path is possible. This becomes particularly distressing when, after having entered an incorrect numerical input, the user has to continue a now meaningless conversation until the computer produces some mendacious results. Moreover, there is no way for the user to speed up the conversation by entering more intricate, composite constructions.

3.3.3 Menu-driven operation

In a menu-driven system, the user performs a series of selections from different menus to control the action taken by the program. This scheme is implementationally similar to a question-and-answer scheme and gives the same advantages, but unfortunately also the same disadvantages. A further disadvantage, also true for question-and-answer systems, is the inflexibility and nonextendability of these systems. If a certain sequence of commands needed to calculate certain results is not foreseen (pre-programmed), it is impossible for the user to get the program to perform the needed action. As an example, let us assume that a main menu lets the user select between discrete-time state-space operations, continuous-time state-space operations as well as other choices. If, for some reason, an eigenvalue computation has been defined only in the sub-menu containing continuous-time operations, there is absolutely no way the user can get the eigenvalues of a discrete-time state-space system, although the necessary operation would be exactly the same as in the continuous case, namely to calculate the eigenvalues of a real square matrix.

A further disadvantage of most menu-driven systems is that the user has no way of combining often used menu-selections into compound commands ("dynamic super-menus"), making it impossible to speed up the normal operation.

The mentioned drawbacks are prohibitive enough, so that no more CACSD-packages purely relying on a menu-driven conversation are implemented. However, menu techniques may still be interesting for CACSD environments. So-called "pull-down" and "pop-up" menus have become an integral feature of modern workstations and this has standardized the user interface to many system and application programs. Their employment may therefore be assumed to be well known to all users of any particular machine. Therefore in different sections of this chapter, we will see how question-and-answer/menu driven operations can be incorporated into other conversational modes to simplify particular types of operations while circumventing the above mentioned drawbacks.

3.3.4 Form-driven input

In this mode of interaction, a predefined form will be displayed on the screen. An example of such a form is given in Figure 3.5. Normally, this form will be partially filled out by the program using default values, enabling the user to supply values for a subset of the data fields of the form only.

Forms are particularly useful when the user has to control a complicated algorithm, construct/scale detailed graphs, and other instances where the program needs specification of numerous parameters, some of which can be assumed to take default (pre-calculated) values. The fields in a form can take numerical, boolean, name or string values. Advanced form-drivers may even change/append the displayed form instantaneously when certain key fields are assigned new values, and this change has to be reflected in some additional fields.

3.3.5 Command-language interaction

This is the most complex among the alphanumeric conversational methods, both for the program implementor and for the user of the program.

The user has to enter commands using a predefined, often quite intricate, command language. This command language may possess a syntax similar to a natural language, a procedural programming language (e.g. Basic or Pascal), or may employ a unique syntax of its own (*Bongulielmi et al., 1985*). Figure 3.6 shows examples of command-languages as used in different CACSD packages. The **L-A-S** example calculates the time-response from an optimal LQ regulator (*West et al., 1985*). The

```
Domain(s)   : OUTPUT_1
Main title : Output from stable system SYS1

X  Lower limit  0       ( 0.000)
   Upper limit  20      (16.000)
   Axis Scale   Linear (Lin/Log/Linrev)
   Axix title   TIME
   Axis unit    h

Y  Variable(s)  V                       VIN
   Axis title   Cable roll-off speed    Desired speed
   Axis unit    m/s                     m/s
   Lower limit  0      ( 0.000)         0      ( 0.000)
   Upper limit  10     ( 8.473)         10     ( 5.000)
   Color        Red                     White
   Line         Solid                   Solid
```

Figure 3.5: Example of a form-driven input for describing a plot-output. In this fictious display, the user should move around with cursor-buttons or a mouse to change individual elements. Parenthesized entries either indicate the possible selections or show the minimum/maximum actual values. Help-information should be available for every field. Note that the screen should be fully dynamically controlled, if for example the user decides to plot yet another variable, another column has to be created or an old one has to be temporarily overwritten.

```
;  Example of L-A-S input
                     ( INP ) = A,B,Q,R
  B, R(INV), B(T) (*) (*) = S
 A, S, A,Q,S(RIC) (*) (-) = AC
                     (INP) = XO
             AC, XO (RCS) = Y
```

```
PROGRAM CC, Version 3
(C) Copyright 1984 by Peter M. Thompson, all rights reserved
HELP = list of commands
CC>@ILICAUS
CC>BUILD & G=4/(S-1.5) + 5*S/(S^2+2*S+5) & QUIT
CC>ILI,CAUSAL
```

```
SYST PROC REG CON
AXES H 0 100 V -1 1
PLOT yr y[proc] u[reg]
STORE yr y[proc] u[reg]
SIMU 0 100
SPLIT 2 1
ASHOW y
SHOW yr
ASHOW u
```

```
// [qs] = CONTROL(a,b)
// CTRL-C Function
[ma,na] = SIZE(a);
[mb,nb] = SIZE(b);
IF ma <> na,      DISPLAY('Non-square A-matrix'), ...
ELSE IF ma <> mb, DISPLAY('Incompatible dimensions'), ...
ELSE qs = b; l = b; FOR i=2:ma, l = a*l; qs = [qs,l];
```

Figure 3.6: Examples of different command-driven CACSD interfaces.

Program CC example shows the user entering a SISO-transfer function to calculate the causal inverse Laplace transform (*CC, 1986*). The **SIMNON** example creates several plots from nonlinear simulation outputs (*Åström, 1985*). The **CTRL-C** exmple shows a small algorithm calculating the controllability matrix of a linear system (*1986*). All the mentioned CACSD packages have copyrights and are listed in ELCS (*1986*).

The programmer implementing a program that is able to accept command language input faces difficulties similar to those found during compiler construction (*Aho et al., 1986*) and/or natural language processing (*Barr and Feigenbaum, 1981, Volume 1*). Moreover, the *interactive* nature of most CACSD packages causes a further problem of attaining an immediate and yet efficient interpretation of the input (see Section 5.6 for a discussion on data- versus code-driven execution).

Command-driven input is normally the fastest and most flexible way of controlling an interactive package. However for the beginner, it is also the most difficult one to master, as any *pure* command-language interpreter gives the user relatively little assistance. With present command-interpreter technology, which is more comparable to programming language compilers than natural language processors, the user must be knowledgeable about a substantial subset of the language syntax before he even can start to use the program. In particular, any incomplete input is normally not acknowledged by the program as a base for further inquiry on the action to be taken, but instead a cryptic message like

```
%FATAL-ERROR FTN-INP-54/66B, Incomplete input.
```

is displayed on the terminal. This reaction can be compared with the situation of a small child learning to talk, while all adults refuse to understand anything except complete sentences! Needless to say, computers have to become much more "human" before they can be called user-friendly. In a later section of this chapter, we will see how other conversational modes can be integrated with command-languages to help the user when he enters fractional or incomplete commands.

Once a command-language has been mastered, the user has access to a very powerful tool. A nontrivial command-languages supporting conditional and structural elements may be used to enter virtually any combination of commands in a highly structured manner. This gives the user the freedom to modify, extend or combine existing commands and, as will be shown later in this thesis, the command interface becomes comparable to any powerful programming language – in fact, we are then talking about special-purpose *interactive programming environments.*

3.3.6 Command-language interaction versus other alphanumeric methods

There is one fundamental difference between these methods – the menu and the question-and-answer methods let the computer be in charge of the conversation, whereas command language interaction gives the user almost total control. A form-driven input lies inbetween these extremes, as the user has full "local control" in that he can modify any of the data fields in arbitrary order, however, he can not extend this control beyond the available, predefined data matrix.

A commonly found combination of the "computer-controlled" methods is to have a menu-driven operation at the outermost level(s), and question-and-answer conversation to obtain more detailed information from the user, once the operation to be performed has been selected from the menu (as in e.g. KEDDC, *Schmid, 1985*).

If the only design goals are a minimum learning time and maximum accessibility by non-specialists, the menu/question-and-answer method is most certainly the correct choice. However, this method gets very tiresome after a while, as the user tends to anticipate the next question, but cannot speed up the input. As an example, again compare the conversational mode of INTOPS in Figure 3.1 with the equivalent MATLAB commands in Figure 3.3. One way to overcome this deficiency is a type-ahead (answer buffer) facility. If each question may be answered with a single symbol only (e.g. a simple name or number) or if a special character for separating answers to different questions has been defined, the user may enter responses to the present question as well as to future, anticipated questions at the same time. Subsequent questions are suppressed until either the answer buffer is exhausted or an incorrect entry has been met in which case the rest of the answer buffer is automatically cleared, and the program returns to its indigenous question-and-answer mode.

The advocates of menu-driven and question-and-answer interaction rightly claim that their methods are specially advantageous for users unfamiliar with the system. In our example, the user of INTOPS needed to know only the existence of the two commands ENTER and INV, whereas the user of MATLAB must know how to form a matrix, and how to enter a variable as parameter of a function. Despite this, due to the very natural notation of the MATLAB command language (an extension of which will be described in more detail later), any inexperienced user will be able to use MATLAB after a few minutes of introduction only.

On the other hand, anyone familiar with both systems will save a factor of 10 in time as well as in number of input lines when he uses MATLAB to compute the inverse modal matrix. This means that the command language interface pays off drastically after not more than just a few hours of use.

One of the most noticeable drawbacks of menu/question-and-answer driven systems is that only pre-programmed sequences of actions can be performed. This is a serious handicap of many CACSD-packages. As no general CACSD-package can include every conceivable control algorithm, especially not if it is to be used as a tool in scientific research, the user must be supplied with an interface flexible enough to let him extend the package according to his own needs. In particular, it must be possible for the user to assemble existing basic algorithms to form more powerful or more general algorithms. This is not possible in a question-and-answer environment. We therefore believe that the primary interface of a modern CACSD package must be command-driven for both speed and flexibility. However, in order to simplify the system for inexperienced users, several types of help-facilities must be made available:

- a regular help-facility giving on-line explanations on available commands and their syntax/parameters,
- a query mode, giving the user the option to switch back

to a question-and-answer dialogue for more complicated commands,

- a form-driven input to which the system automatically turns whenever a selected operation requires specification of numerous, but individually simple, parameters.

3.4 The command-driven interface of MATLAB

One of the first persons to realize the importance of an interactive command-language interface to packages containing reliable mathematical algorithms was Cleve Moler (*1980*). In his program MATLAB, a milestone in the history of interactive programs, an easy-to-use interactive interface was provided to the LINPACK and EISPACK matrix manipulation libraries. Using a very natural input command language, MATLAB allows to perform matrix operations with the same ease as one executes scalar computations on a pocket calculator.

MATLAB is extremely easy to use, and moreover, MATLAB is very versatile in that new algorithms can be interactively defined using the MATLAB command language. However, the software was never intended to be used in control theory, and therefore, many control problems are not solvable using MATLAB. One reason for this is of course the lack of suitable algorithms, another, more fundamental cause is the lack of appropriate data structures.

As MATLAB is written in a very well-structured manner, it is quite simple to add new algorithms. MATRIX$_X$ (*Walker et al., 1982*) and CTRL-C (*Little et al., 1984*) are well-known examples of control-oriented code-extensions to MATLAB. Although these new products are definite upgrades of MATLAB, they only partially provide the control engineer with an adequate tool. The major reason for the continuing deficiencies of these systems is the above mentioned lack of adequate data structures adapted to control problems. MATLAB uses the complex double-precision matrix (with the scalar as a special case) as its only data structure. All MATLAB upgrades added new algorithms to the software, but left the available data structures basically untouched. However, control engineers often work with more intricate data structures such as polynomial matrices, transfer-function matrices, and linear as well as nonlinear system descriptions.

Furthermore, although the input command language of MATLAB is well suited for smaller problems, a better structured command language is needed for more complicated problems. In particular, a flexible macro/procedure facility accepting parameters must be supported. This feature is partially implemented in MATRIX$_X$ and CTRL-C, whereas it is not available in most other code-extensions of MATLAB.

Finally, versatile graphical output facilities and an interface to a data base should be present. Some of the MATLAB upgrades provide for appealing graphical output, whereas they are all chronically weak it terms of data maintenance.

While MATLAB was implemented in FORTRAN, a few newer CACSD-packages have been implemented in better structured and modular programming languages. Thereby, more intricate data structures can be supported, and modern software engineering principles could be used during the implementation. Some packages that are conceptually based upon MATLAB, but were re-implemented in another language are PC-MATLAB (implemented in C; *Moler et al., 1985*), EAGLES/M (implemented in Objective-C; *1986*) and IMPACT (implemented in Ada, described in the following).

3.5 IMPACT

The rest of this chapter will discuss some important user communication features of modern CACSD programs. Unless otherwise indicated, all examples will be given using the command-format of IMPACT, the CACSD package designed and implemented during the project described in this thesis. A full description of IMPACT will be found in an upcoming User's Manual, an early version of IMPACT has been previously documented by Rimvall (*1983*).

At a superficial glance, the user-interface of IMPACT might appear to be just another extension to that of MATLAB. However, seen from an implementational view, IMPACT is only a conceptual superset of MATLAB. As IMPACT is implemented in Ada, not one single line of code has been taken from MATLAB. Furthermore, several new data structures are supported (discussed in Chapter 4) and the user-interface is far more flexible than the single-mode command language of MATLAB.

IMPACT has been designed with the particular objective in mind to serve a very inhomogeneous group of users. On the one hand, IMPACT is aimed at being used by students with little experience in control theory and no experience at all in CAD. Using only the most basic structures of the input command language which are simple enough to be mastered in a few hours, these students will be able to access intricate algorithms in order to solve demanding control problems with a minimum amount of tutorial assistance. An on-line `HELP` facility contains all necessary information on the command language syntax as well as on the numerical algorithms, making self-tutorial possible. A query-feature guides the user through calls of involved, multi-parameter functions.

On the other hand, the experienced control engineer is provided with a full-fledged structured command language containing all elements found in a high-level computer language including `WHILE` and `FOR` loops, `IF-THEN-ELSE` statements, and so forth. Furthermore, a large selection of IMPACT functions and procedures gives the user access to a wide range of algorithms. To further enhance the structurability, a number of different macro facilities have been introduced.

3.6 The basic command-language of IMPACT

The process of *designing* an interactive command language is similar to that of formulating a new computer language. Consequently, it is imperative that a formal notation for the syntax (grammar) is used, and that this language representation is tested by some automatic syntax-checker (*Bongulielmi et al., 1984*). Only then can the consistency and completeness of the new language be guaranteed. Thus, all language elements of IMPACT have been designed and tested using the general-purpose parser (syntax-checker) of Bongulielmi and Cellier (*1985*). This syntax-checker accepts languages adhering to the LL(1) class of grammars. Please, refer to Appendix 1 for the complete LL(1)-syntax of IMPACT.

In the *implementation* phase, programs to read, decode, interpret, and execute user input have to be constructed. Here, the relative simplicity of the LL(1) grammar leads to a fairly straight-forward implementation if the implementation language allows for recursive entry (as in the case of Ada but not of FORTRAN). This will be shown in Chapter 5.6.

As most commands entered by a user during a normal session are quite simple, the grammar implementing such typical operations has been given a very simple-to-use syntax. Some valid IMPACT statements for entering data are

```
A = [1,2,0
     0,2,3
     0,0,3]
```

and

```
[9;8;7]
```

for entering a 3*3 matrix with the name `A` and a column vector with the default name `ANSWER`. Similarly as in MATLAB, the thus constructed variables can be used in more involved expressions such as

```
B = [A,ANSWER
     1,2,3,4]

C = [ANSWER;6]
```

to form a 4*4 matrix and a 4*1 column vector. The expression

```
B\C
```

will solve an equation system and store the result again under the name ANSWER, thereby overriding the previous value of this variable. Note that the latter statement is not consistent with an LL(1) grammar: while the identifier B is parsed, it is not clear if this marks the beginning of an explicit assignment statement such as

```
B = A * ANSWER
```

or, as in our case, the start of an implicit assignment to ANSWER. The only other non-LL(1) construction is the procedure call, where during parsing, it is not clear if a non-reserved word refers to a variable/function included in an expression, or a procedure name such as

```
LOAD("myfile.macros")
```

These (at least semantically) ambiguous language elements are solved by means of delayed interpretation in the construction of threaded code, and semantical analysis of possible procedure-names, respectively. Although language-theoretically unpure, this solution seemed justified as the alternative would have been a much more complicated syntax for these common operations.

Most expressional language elements of MATLAB can be found in IMPACT as well. Two examples are the selection of entire rows (columns) from a matrix, and element-by-element operations:

```
G = B(1,:);

H = G .* C
```

Other expressional constructions have been added to describe the more intricate data structures available in IMPACT (see Chapter 4).

3.7 Structured language elements in IMPACT

While an LL(1)-consistent version of the MATLAB interface as described in the previous section is sufficiently rich for forming assignments and simple operations, the flow-control statements of MATLAB are neither rich nor structured enough for a versatile CACSD package. The user must be able to use such statements to interactively define new or adapted control algorithms. Thereby, he must have a tool powerful enough to let him combine predefined primitives in a structured manner. On the other hand if the command language is made too rich, the complexity of the system makes it hard to use.

Thus, the designer has to make compromises in the design of the complete command language, and thereby consider the following aspects:

- Taking the development in software engineering during the last decade into account, the designed language should be highly structured and contain flow control elements like `FOR` loops, `WHILE` loops, and `IF` statements.

- The command language could be developed from scratch, giving the developer full freedom of design, or it could be derived from any existing structured computer language such as Algol, Pascal, or Ada. The latter approach will decrease the learning time for all users familiar with that particular programming language.

- Together with the available data structures, the command language must be made rich enough to describe control algorithm. Therefore, if an existing programming language is taken as a starting point, some extra language elements for handling the available data-structures must be added. Other language constructions, such as involved variable declarations (strong typing) and general input/output operations may be omitted.

- As the user input, including all functions described in the command language, has to be interpreted rather than compiled, the execution of long algorithms described in the command language must be rather slow. During the development of new algorithms, this is offset by the time not spent on compilations. However, to obtain shorter execution times, algorithms should be compilable, and it should be possible to incorporate once compiled algorithms into the package itself as soon as they are completely developed and tested. If the command language is made similar to the implementation language, such a transition can eventually be automated, or at least partially automated involving a minimum of manual re-coding.

Considering these aspects, a command language resembling that of the Ada programming language has been developed for IMPACT (see Appendix 1). Thereby, the necessary structured flow-control elements are identical to those of Ada. We will illustrate this with two examples:

Example 1: The heat-diffusion in a long metal bar isolated on one side can be approximated by a set of `N` differential equations (u is the temperature of a heat sink at the unisolated side):

$$\begin{aligned}
\frac{dtemp(1)}{dt} &= (-2 * t(1) + t(2) + u) * k \\
\frac{dtemp(N)}{dt} &= (-2 * t(N) + 2 * t(N-1)) * k \\
\frac{dtemp(i)}{dt} &= (-2 * t(i) + t(i-1) + t(i+1)) * k \ , \ 1 < i < N
\end{aligned}$$

where

$$k = \frac{hcoeff}{deltx^2}$$

In Figure 3.7 we show how the state- and input-matrices of this model can be obtained through a nested set of `FOR` loops and `IF` statements. Note the similarity to the Ada syntax.

As many matrix- and control-algorithms contain operations performed on each individual element of matrix structures etc., a shorter notation of the above nested loop can be obtained with a `FOR INDEX` statement, as shown in Figure 3.8. This unique lan-

```
n = 25;
FOR i IN 1 .. n LOOP
  FOR j IN 1 .. n LOOP
    IF (j = i) THEN a(i,j) = -2;
    ELSIF abs(j-i)=1 THEN
      IF (i = n) THEN a(i,j) = 2;
                 ELSE a(i,j) = 1;
      END IF;
      ELSE a(i,j) = 0;
    END IF;
  END LOOP;
 IF (i = 1) THEN b(i) = 1;
            ELSE b(i) = 0;
END LOOP;
```

Figure 3.7: Creating the state- and input-matrices of the metal-rod model using nested loops

```
n = 25;
a = ZEROS(25);
FOR INDEX IN a(i,j) LOOP
  IF (j = i) THEN a(i,j) = -2;
  ELSIF abs(j-i)=1 THEN
    IF (i = n) THEN a(i,j) = 2;
               ELSE a(i,j) = 1;
    END IF;
   ELSE a(i,j) = 0;
  END IF;
 IF (i = 1) THEN b(i) = 1;
            ELSE b(i) = 0;
END LOOP;
```

Figure 3.8: Creating the state- and input-matrices of the metal-rod model using the indexed loop construction

```
n = 25;
z = ZEROS(n-1,1);
a = -2*EYE(n) + [z,EYE(n-1);0,z'] + ...
                [z',0;EYE(n-1),0];
a(n,n-1) = 2;
b = [1;z];
DELETE(z)
```

Figure 3.9: Creation of the state- and input-matrices of the metal-rod model using connected matrix constructions

guage element of IMPACT create nested loops with incremental or decremental counting of the indices. Thus, the statement

```
FOR INDEX IN a(REVERSE i, REVERSE j) LOOP
```

will decrement each counter from N (M) to 1 with i in the "outer loop" and the statement

```
FOR REVERSE INDEX IN a(i,j) LOOP
```

will put j in the "outer loop".

Although these structured flow-control statements are necessary for programming more intricate algorithms, even in this example, we can use the strong expression-power of IMPACT illustrated in Figure 3.9 (using MATLAB-related constructions), and thereby shorten the input compared to the previous FOR loop based constructions considerably

Example 2: Let us consider the problem of solving the Riccati Equation

$$\begin{aligned} \dot{x} &= A * x + B * u \\ y &= C * x \end{aligned}$$

where

$$\int_0^\infty (x' * Q * x + u' * R * u)dt \overset{!}{=} MIN$$

```
a = [...]; b = [...]; q = [...]; r = [...];

<v,d> = EIG([a, -b*(r\b'); -q, -a']);
k=0; n = DIM(a);
FOR j IN 1 .. 2*n LOOP
  IF REAL(d(j,j)) < 0 THEN
    k=k+1;
    v(:,k) = v(:,j);
  END IF;
END LOOP;
p  = REAL(v(n+1..2*n,1..k)/v(1..n;1..k));
fc = -r\b'*p
```

Figure 3.10: Interactive commands for solving the Riccati equation over the Hamiltonian matrix

After defining the matrices A, B, Q and R, this problem can be solved by a simple algorithm first described by Potter (*1966*). The IMPACT solution is shown in Figure 3.10. In this algorithm, we first compute the eigenvectors and eigenvalues of the Hamiltonian. We store the eigenvalues diagonally in `d` and the eigenvectors as columns in `v`. Thereafter, we extract those columns of `v` that correspond to negative eigenvalues, cut this reduced eigenspace into an upper an a lower part, compute the stable and positive definite Riccati matrix `p` as the solution of a linear system, and finally evaluate from there the feedback coefficients `fc`. We note the similarity with Ada, the main differences originate in the notation of MATLAB and include the use of = for assignment statements , [and] to describe the mathematical matrix structures, and : to form substructures (e.g. to form column vectors out of matrices). < and > are used to delimit the output variables of a multiple-output function such as `EIG`.

3.8 Subprograms

In the previous section, we have seen examples of how sequences of basic statements can be put together to form control algorithms. However, the system could not be efficiently utilized if the user were forced to enter all these statements anew each time he wanted to compute a Riccati feedback. Therefore, IMPACT supports function and procedure subprogram (macro) facilities, permitting the user to describe entities (sequences of action) once, and then use them repetitively. These user-defined subprograms may thereafter be called using exactly the same syntax as for the standard, Ada-coded subprograms that are intrinsic parts of the IMPACT system (such as `EIG` and `REAL` in the previous example). In the following, we will call Ada-coded subprograms *hard-coded*, and subprograms defined using the interactive command-language of IMPACT *soft-coded.* Note that soft-coded is not synonymous to user-coded. Many of the standard IMPACT subprograms are actually soft-coded as well. Inversely, the user may add hard-coded routines to his own version of IMPACT.

IMPACT also supports another kind of macros, called system macros, to describe nonlinear systems. Although the syntax of these macros is similar to that of functions and procedures, we will postpone their discussion to the next chapter on data structures.

3.8.1 Function subprograms

As in conventional programming languages, the functions of IMPACT may be used freely within expressions to return a value calculated from the given function parameters. However, just as MATLAB, IMPACT supports functions returning more than one value per call. The following example illustrates this:

If the previously described model of a metal bar is to be used several times with different `N`'s, the user can save time by defining a function macro returning the wanted matrices. This function definition is shown in Figure 3.11. We note the similarity to standard Ada function declarations. The characters $<$ and $>$

```
FUNCTION Bar_matrix(n) IS
BEGIN
  z = ZEROS(n-1,1);
  a = -2*EYE(n) + [z,EYE(n-1);0,z'] + ...
                  [z',0;EYE(n-1),0];
  a(n,n-1) = 2;
  b = [1;z];
  RETURN <a,b>;
END Bar_matrix;
```

Figure 3.11: Definition of a function to create the the state- and input-matrices of the previously defined metal-rod model

are used in the RETURN statement as well as for the multiple assignments to delimit the return-list. The thus defined function can then be called as

```
<aa,bb> = Bar_matrix(25);
```

Also the Riccati-example of the last section can be defined as a function. This is shown in Figure 3.12. Note that no declarative typing of the parameters was made, and local variables were created without being previously declared. The reason for this disparity to the strong typing of Ada is that, in IMPACT, command sequences ("algorithms") may be entered interactively. It would then be very cumbersome for the user to be forced to define every single variable in the beginning of every session, especially if he does not exactly know which method to use and which intermediate variables he will need. On the other hand, explicit variable declarations help detecting programming errors in functions and increases the security of the functions by performing run-time type checking on the parameters. Therefore, as illustrated in Figure 3.13 the header of the Riccati function could be complemented with type declarations in which case *all* variables used in the function must be declared. After the Riccati function has been defined, we can access it as a normal function

```
Function Riccati(a,b,q,r) IS
BEGIN -- Riccati
  <v,d> = EIG(<a, -b*(r\b'); -q, -a'>);
  ..
  ..
  p = REAL(v(n+1..2*n,1..k)/v(1..n;1..k));
  RETURN -r\b'*p;
END Riccati;
```

Figure 3.12: Definition of a function to solve the Riccati equation

```
FUNCTION Riccati(a,b,q,r : MATRIX)
                 RETURN MATRIX IS
  v,d,p : MATRIX;
  k,n   : INTEGER;
  -- j is an implicitly declared loop counter
  -- (cf. Ada)
BEGIN -- Riccati
  ..
```

Figure 3.13: Definition of a function solving the Riccati equation with parameter and variable type declarations

```
FUNCTION Riccati(a, b : MATRIX;
                 q    : MATRIX = EYE(DIM(a)),
                 r    : MATRIX = EYE(DIM(b,2)))
                 RETURN MATRIX IS
  v,d,p : MATRIX;
  ..
```

Figure 3.14: Definition of a function solving the Riccati equation with defaulted parameter values for shortened call sequences

```
ffcc = Riccati(aa,bb,qq,rr)
```

It is currently foreseen that explicit declarations will be made mandatory for functions to be automatically pre-compiled into Ada for inclusion into the set of hard-coded IMPACT functions.

Characteristic for many control algorithms is that they rarely appear in one version only. Often, this multitude of versions is caused by structural variations in the treated systems. Other times, these structural variations do not influence the algorithm itself, but lead to particular parameter selections (affecting error tolerances, type of input parameters, and so on).

In IMPACT, the combination of defaulted formal parameters and named rather than positional actual parameters can be used to take care of both algorithmic and parametric variations. For example in our Riccati equations, the `Q` and `R` matrices are often chosen to be unity matrices. In IMPACT, we could shorten many calls to Riccati by defining default values as shown in Figure 3.14. We have in this figure defined `Q` and `R` to be square unity matrices of correct dimensions. We make use of the fact that the actual assignment of default values is performed at the time of call, when the dimensions of the `A` and `B` matrices are known. A default value will only be used when no other value is specified for the parameter in question.

After this re-declaration of function Riccati, it can be called with a shortened parameter list For example, if `Q` is a unity matrix, we would call Riccati as

```
    ffcc = Riccati(aa,bb,,rr)
```

or, a little more readable

```
    ffcc = Riccati(aa,bb,R=>rr)
```

As in Ada, it is always possible to specify named rather than positional parameters (useful for documentation to increase clarity):

```
    ffcc = Riccati(A=>aa,B=>bb,R=>rr)
```

Similar to the hitherto examples with defaulted unity-matrix parameters, algorithms requiring error-tolerances are often defined with default tolerances which can be overridden by the user whenever needed. The function for pseudoinverse therefore uses the machine tolerance as an annullable default:

```
    pia = PINV(a);
```

but we can at any time override this default by an alternative value:

```
    pia = PINV(a,TOL=>1.e-5);
```

3.8.2 Procedure subprograms

The parameters of an IMPACT function were all IN-parameters, that is, they could be accessed but never assigned a new value within the function body. Conversely, procedures may have IN, OUT and IN-OUT parameters (with obvious semantic meaning). Hence, the computation of the state-matrices for our metal bar could be made into the procedure of Figure 3.15 and called as

```
    Bar_matrix(aa,bb,25);
```

Some hard-coded procedures have a variable-length implicit parameter declaration which lets the parser adapt itself from call to call. As an example, Figure 3.16 shows that the procedure DELETE can take any number of parameters, each of which is the name of a variable to be deleted. Moreover, certain string parameters change the meaning of the procedure completely.

```
PROCEDURE Bar_matrix(a, b :    OUT MATRIX;
                     n    : IN     INTEGER) IS
  z : MATRIX;
BEGIN
  z = ZEROS(n-1,1);
  a = -2*EYE(n) + [z,EYE(n-1);0,z'] + ...
                  [z',0;EYE(n-1),0];
  a(n,n-1) = 2;
  b = [1;z];
END Bar_matrix;
```

Figure 3.15: Definition of a procedure which creates the the state- and input-matrices of the previously defined metal-rod model

```
DELETE(pia)                    -- deletes the variable pia
DELETE(a,b,r,q)                -- deletes a,b,r,q
DELETE("all")                  -- deletes all variables
DELETE("all",TYPE=>"function")-- deletes all user-defined
                               -- soft-coded functions.
                               -- System-defined subprograms
                               -- cannot be deleted by a user.
```

Figure 3.16: Overloaded use of the procedure DELETE. This procedure takes a variable number of parameters of different types

3.9 The use of parallel sessions

Anyone remembering the times when research still was made using pencil and paper can certify the advantages of using two or more sheets of paper concurrently. On one sheet, the actual results (e.g. theorems and proofs) were painstakingly noted down. On the other sheet(s), intermediate computations were performed, alternative paths were examined, numerical examples were tested, and so on. When the investigation was completed, the scratch-sheet(s) were discarded, leaving only the clean results. Then the computer became widespread and you started to write large FORTRAN programs, possibly using subroutines from standard algorithmic libraries. As award for and proof of your strenuous programming work, you received the results (together with program listings, memory dumps and intermediate results) on over-sized sheets of papers which did not fit into any loose-leaf binder, turning your book-shelves into paper-shelves. Then, suddenly, you got your first interactive CACSD package. You could now work on your CRT-terminal, and all intermediate results appeared only on the terminal screen. At the end of your session, you saved only the main results on an external file to be printed, made an exit from the program and all intermediate results disappeared.

Indisputably, these new interactive CACSD-packages have become invaluable tools in modern control research and development. However, they have not yet brought back the comfort of your scratch-sheet. You normally work interactively with a single set of variables, all residing in "one big bucket". After an hour or so of work, you have most certainly lost the overview of your variables, especially since you may be forced to give each of them a meaningless, four-letter name. The only remedy is to delete every intermediate variable (giving you the function of an eraser instead of a scratch-paper).

The man-machine interface of IMPACT avoids this accumulation of variables, and gives the user his scratch paper back by introducing a concept of *sessions*. Each session is a logical work-area, conceptually comparable to the directory structures of many operating systems (REF) and to the windowing techniques of modern workstations like the VAX-station II, Apollo

Domain, and Sun workstations. It is possible to open several sessions during one call to IMPACT. These sessions are then concurrent. The user can switch between sessions at any time, and each session is completely independent of the others. In each of these sessions, the user can define local variables, guaranteeing that the number of current variables in each session is kept to a minimum, and that related data structures are kept together. Moreover, the session feature does not add any unreasonable complexity to the basic IMPACT system described in the previous sections. In fact, the novice user can ignore the session facility to start with, and begin structuring his work-environment only as he gets more familiar with the system.

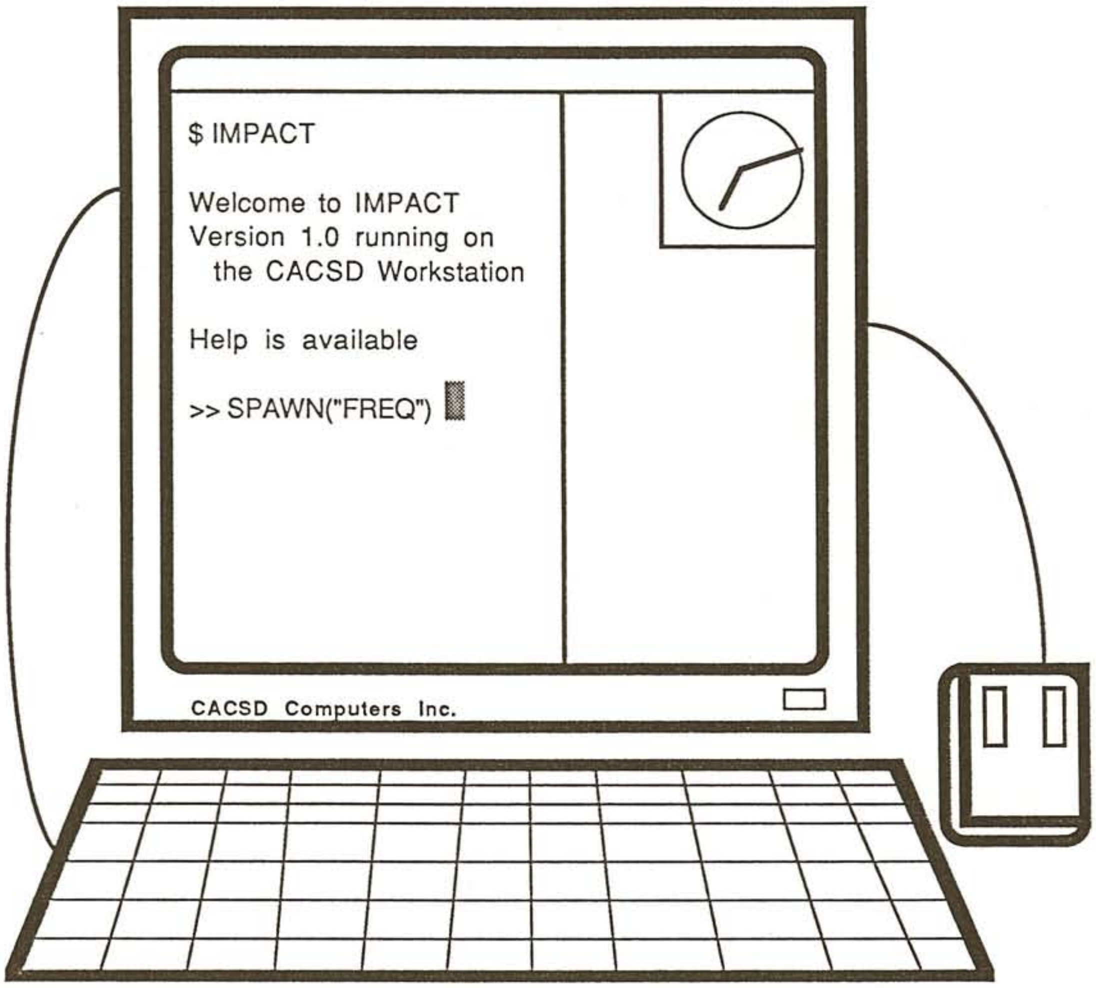
To get an intuitive feel for how sessions can be used, let us consider a user working on a control problem in both the frequency and the time domain. Whenever IMPACT is started on a modern workstation, a large window will be opened, and a welcoming message will be displayed together with the prompt of the main session. Figure 3.17 illustrates this on a fictitious workstation. The user may now decide to form three additional sessions: FREQ, TIME and SCRATCH. The first of these new sessions is created with the command

```
>> SPAWN("FREQ")
```

which will propel the system to open another window, see Figure 3.18 On this new window, the prompt FREQ>> is displayed. The user can now proceed to create (open) the sessions TIME and SCRATCH. Thereafter, he can either use the mouse to click on different windows (see Figure 3.19), or alternatively, he may use the command SWITCH:

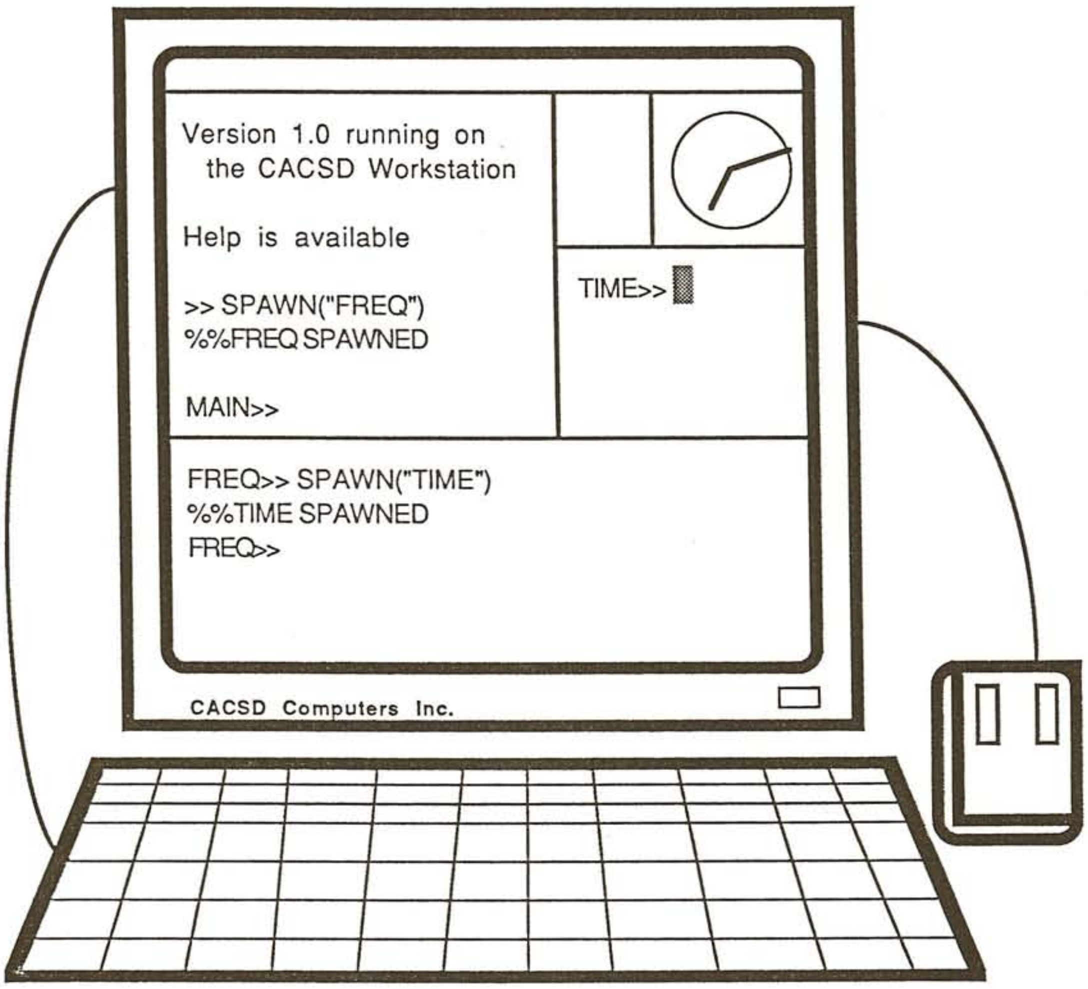
```
SCRA>> SWITCH("FREQ")
```

All sessions are logically non-hierarchical, they are ordered alphabetically according to their names and given equal priority. If a user closes a session without an indication to which session he wishes to transfer, he will find himself in the MAIN session, or, if the main session has been deleted as well, in the alphabetically first session.



```
$ IMPACT

Welcome to IMPACT
Version 1.0 running on
  the CACSD Workstation

Help is available

>> SPAWN("FREQ")
```

Figure 3.17: Workstation where the main session has been started.

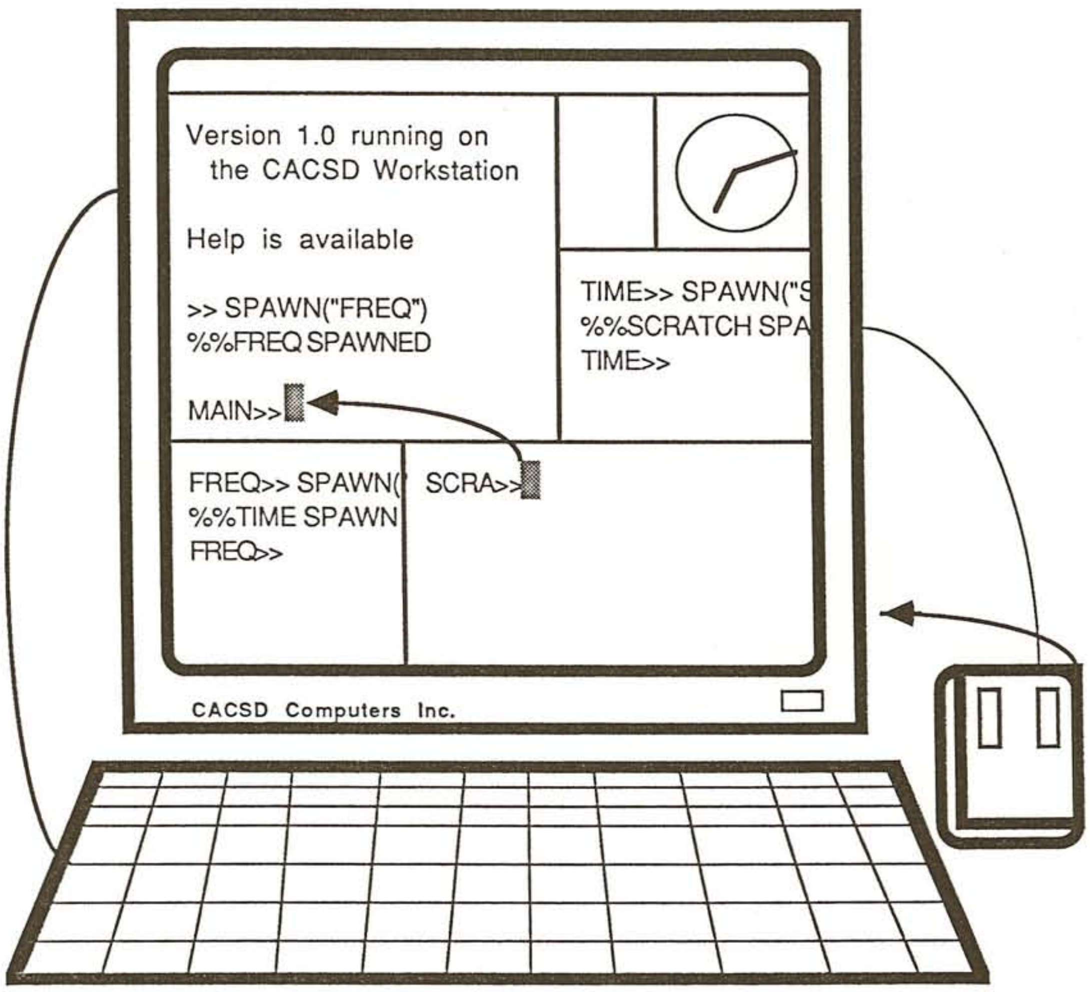


```
Version 1.0 running on
  the CACSD Workstation

Help is available

>> SPAWN("FREQ")
%%FREQ SPAWNED

MAIN>>
```

```
TIME>>
```

```
FREQ>> SPAWN("TIME")
%%TIME SPAWNED
FREQ>>
```

Figure 3.18: Workstation where the session FREQ has been opened.



```
Version 1.0 running on
  the CACSD Workstation

Help is available

>> SPAWN("FREQ")
%%FREQ SPAWNED

MAIN>>
```

```
TIME>> SPAWN("S
%%SCRATCH SPA
TIME>>
```

```
FREQ>> SPAWN(
%%TIME SPAWN
FREQ>>
```

```
SCRA>>
```

Figure 3.19: Switching between different session using mouse movement.

On a normal, alphanumeric terminal not supporting windowing techniques, the logical sessions can not be displayed concurrently as different areas on the screen. However, the SPAWN and SWITCH commands are still available, and the prompt used will remind the user to which session he is currently attached. This corresponds to the present implementation. In a later update, concurrent sessions will be handled by the tasking facility of Ada, guaranteeing a fairly portable implementation despite the heavy hardware/firmware host-dependency of any windowing techniques.

We have discussed how a problem can be decomposed into several subproblems, and how each of these subproblems can be treated in separate sessions to avoid large accumulations of variables. However, sometimes the user needs a variable from another session. Such a variable can be directly accessed using a dot-/colon-notation. For example, if the user wants to work in the frequency domain (session FREQ) on the transfer-function equivalent of a linear system description sys1 defined in the time domain (session TIME), he can access and transform this system directly (TRANS transforms a linear system description to a transfer-function) by

```
FREQ>> sys2 = TRANS(time:sys1)
```

or, alternatively, within session TIME:

```
TIME>> freq:sys2 = TRANS(sys1)
```

In the case where a new scratch session is started for some intermediate computations, it is sometimes meaningful to import some or all variables from another session:

```
SCRA>> IMPORT(freq:,"all")
```

The colon indicates that freq is a session-name. Without this colon, IMPACT would look for a string-variable with the name freq. Alternatively, we may supply the procedure with a string containing the name of the session:

```
SCRA>> IMPORT("freq","all")
```

Yet another alternative would be to create a string with the name of the session from which all variables are to be imported, and pass this string as a parameter to the procedure IMPORT:

```
SCRA>> dd = "freq"
SCRA>> IMPORT(dd,"all")
```

Note how a high flexibility of the use of procedure IMPORT is attained through this overloading of allowed parameter types.

The command DELETE can be used to close active sessions and delete any local variables. If the current session is deleted, the user will find himself in the MAIN session:

```
SCRA>> DELETE("scratch")

>>
```

3.10 The query feature

With the data structures and command language of IMPACT, the advanced control engineer is given a very powerful algorithmic environment which he can further adapt to his own needs. On the other hand, if first time users are directly presented with the *full* IMPACT package, they will most certainly be stunned by its complexity. Many CACSD-packages try to resolve this problem by including an interactive HELP facility. However, while such help is excellent once a user has acquired a general overview of the package and only needs information on a particular subject, to a novice user, it is as pedagogic as the index of a 200 page reference manual. Of course, IMPACT does support an interactive HELP facility, but to prevent the initial shock, it also gives the user a gradual introduction. A tutorial will be made available that presents the most simple language elements, e.g. how to create variables and how to call standard functions. Moreover, as even this might be too complicated for a beginner unfamiliar with the standard concepts of control theory, a

```
TIME>> FF = Riccati(HELP)
RICC>>The function Riccati solves a Riccati equation.
RICC>>A Riccati equation is defined by the A and B
RICC>>matrices of a linear systems and an
RICC>>integral involving two weighting matrices
RICC>>Q and R (often unity matrices) where
RICC>>
RICC>>.              /inf                           !
RICC>>x=A*x+B*u  and  |    (x'*Q*x + u'*R*u)dt = MIN
RICC>>               /0
RICC>>
RICC>>System matrix A (NO DEFAULT): [0,1;0,-1]
RICC>>Input matrix  B (NO DEFAULT): [0;1]
RICC>>Weight matrix Q (DEF=Unity) : [1,0;0,0]
RICC>>Weight matrix R (DEF=Unity) :

FF =
   1    0.7321
```

Figure 3.20: Invoking the query feature by including the standard parameter HELP in a function call

query-mode has been introduced. In this mode, the initiative is transferred from the user to the system. Through a guided question-and-answer conversation, the system will determine the correct action to take.

Assuming that an inexperienced user wants to use the for him new function Riccati. As illustrated in Figure 3.20 he would then call the function using the HELP qualifier, forcing IMPACT to enter the query mode. Thereafter the user will be asked to supply values values for each of the parameter. Optional (defaulted) parameters need to be specified only when another than the default value is to be used.

Especially for functions with many parameters, this facility is very useful. Moreover, if the user is uncertain about the meaning of a particular parameter, he can enter HELP for further information on that query-request, as illustrated in Figure 3.21. If at this point the users has forgotten the name of an already created system matrix, or if he has to perform some intermedi-

```
RICC>>System matrix A (NO DEFAULT): HELP
RICC>>Please enter a matrix describing the state
RICC>>connections of a linear system. This matrix
RICC>>is the A matrix in the system description
RICC>> .
RICC>> x=A*x+B*u
RICC>> y=C*x+D*u
RICC>>
RICC>>System matrix A (NO DEFAULT):
```

Figure 3.21: Requesting additional help on individual parameters of a queried function call

ate computations, one of the options available to him would be to open yet another interactive session through the command SPAWN, and thereby get access to the general help facility, the command DIRECTORY or any other IMPACT statements. He can thereafter return the generated state matrix directly to the Riccati query function:

```
RICC>>System matrix A (NO DEFAULT): SPAWN
```

This will start a new session (open a new window) with the name RICC_1 (only the first four letters of the function-name are retained) which automatically imports all the variables from the currently active session.

```
RICC_1>> HELP
 ..
RICC_1>> A = A_MAT(sys1); -- extract A from sys1
RICC_1>> RETURN A
```

Return to RICC and continue the query with the next parameter.

```
RICC>>Input matrix  B (NO DEFAULT):
..
```

The user has three choices to close session `RICC_1`: With `RETURN 'value'`, he deletes the intermediate session `RICC_1`, and at the same time returns a value to the query-function. Newly created or modified variables of the session `RICC_1` will thereby be destroyed, and will not be exported back into the currently active session. The commands `RETURN` without parameters or `DELETE(RICC_1:)` both close `RICC_1` and force IMPACT to return to the original query for `A`.

`RICC_1` is a subsession of session `TIME`. From `RICC_1` the user may `SWITCH` to any other session and perform any other actions. However, the original session `TIME` is blocked and can not be `SWITCH`ed to until session `RICC_1` is closed (deleted). If the user (recursively) calls function `RICCATI` from `RICC_1` and again `SPAWN`s out of the query-mode, `RICC_1` is blocked and a further subsession `RICC_2` is created.

To tune the package to the level of proficiency of the individual user, IMPACT can be set by the user to activate the query facility on three different trigger-levels:

- On the lowest and most comprehensive trigger-level, the query-facility will be activated as soon as any subprogram is called without a complete set of parameters. In particular, the query-facility will be invoked also when only parameters with default-specifications are missing from a parameter-list. This trigger-level is intended for the user wishing to keep himself informed on every detail including defaulted parameter values of the called algorithms.
- On the normal level, the query-facility is invoked whenever a non-defaulted parameter is missing. Once invoked, the user will be asked to supply values for all parameters, including those having default values. This should be ideal for most users during normal operation.
- The highest trigger-level is intended for the experienced user wishing to receive an error message each time he enters an incomplete statement. On this level, there is no automatic query for missing parameters. However, the user can activate the query for individual subprogram calls by including the parameter HELP in the parameter-list.

The high trigger-level is always used within soft-coded subprograms, as unexpected queries caused by incomplete subprogram calls from within another subprogram may get very confusing to the user. However, it is possible to invoke the query-facility from within a soft-coded subprogram body through a HELP parameter. It is recommended to employ this possibility sparcely, and only in combination with adequate clarifying messages to the user.

To economize in the time needed to formulate and type the content of the three main textual information aggregations of IMPACT (HELP- and QUERY-texts as well as sections of the USER'S MANUAL), a single textual source is maintained for all three descriptions of subprograms (cf. also Section 3.11). During the automatic processing of this text, most of the information is sent to all three information aggregations, but it is possible to limit some information to one or two destinations, for example to place a more extensive example only in the off-line manual.

3.11 Query mode and soft-coded subprograms

In the last section, we saw an example of how the query option can be used in connection with the system maintained Riccati function. We now have to ask ourselves the following obvious question: How does IMPACT know what information to give the user when HELP is requested for a function that was soft-coded by the user just a few minutes ago? Well, the answer is as obvious. Either IMPACT does not know, in which case the query-mode only prompts for the parameters using the formal parameter names (which is not much help for a person not familiar with the function), or we have told IMPACT what to ask. The definition of our soft-coded Riccati function is therefore extended in Figure 3.22 once again

The syntax for parameter-/help-information is identical for hard-coded subprograms. Each of these subprograms also has a "soft-coded" definition for the parameter- and help-information. Therefore, the same routines can be used for the parameter-

```
FUNCTION Riccati(a, b : MATRIX;
                 q    : MATRIX = EYE(DIM(a)),
                 r    : MATRIX = EYE(DIM(b,2)))
                 RETURN MATRIX IS
  ?BEGIN?
  ?GI? The function Riccati solves ...
       A Riccati equation is       ...
  ...
  ?PN? A
  ?PS? System matrix A (NO DEFAULT)
  ?PI? Please enter a matrix       ...
       connections of a linear     ...
  ...
  ?END?
  v,d,p : MATRIX;
  k,n   : INTEGER;
BEGIN -- Riccati
  ...
  ...
  RETURN -r\b'*p;
END Riccati;
```

Figure 3.22: Including query-information in a soft-coded subprogram

handler and the query-controller for both soft- and hard-coded subprograms.

3.12 Data-base interfaces in CACSD

Although large, integrated CACSD packages may be powerful and versatile enough to serve the user during all phases of the control cycle (see Sections 2.2 and 2.3), there are still several instances where a communication with the outside external world is needed, for example

- to store away or reenter individual sets of data or functions. This could for example be used to save the description of a system or the definition of a soft-coded function for later use.
- when it is necessary to exchange data with other programs or processes. This is for example needed when measured data are to be entered, or when the CACSD package does not include any simulation facility, and a connection with a nonlinear simulation package is needed.
- when the user wishes to exit from the interactive program, but intends to resume operation at a later time. It is then necessary to store away the complete interactive context (including several sessions). It must thereby be possible to reenter the whole context, so that operations can be resumed exactly where the user left off.

Apart from saving the total context, which must be seen as a single save-set deposit to be read as a whole again (and for efficiency usually is implemented as a crude memory dump and memory fill), the data may be stored within the file-handling system of the computer in several manners:

- Each variable or function may be stored in a single external file.
- The user may group several variables and/or functions together and put them into one external file. This allows the user to accumulate related data or subprograms (or both) into libraries.

- In the most general case, a complete data-base facility for storing data externally is supported. This facility should allow the user to group the data together in any fashion he chooses, for example by collecting all data belonging to a single problem into hierarchically organized groups. Such a system should also provide tools with which the user can search a large data-base and collect data or subprograms with a certain name, structure or dimension. An automatic load facility which will search one or more databanks for internally undefined subprograms would also enhance the flexibility of the package.

The difficulties of implementing a data-base interface for a CACSD package increases with both the complexity of the data to be stored away, and the flexibility in which this data is to be referenced in the external storage. The internal representation of control oriented data structures is in itself difficult (these problems are discussed in greater detail in Chapters 4 and 5.4 as well as by Maciejowski, (*1984*). When this same control oriented data is to be stored externally for later reference by either the same or a different program, we are confronted with the even more serious difficulty of storing this structured data efficiently as well as in a portable manner. Portability problems occur on the one hand when the same program is running on different machines under different operating systems, and on the other hand when different packages are to communicate with each other. The only machine-portable solution is to use a sequential ASCII-file. This, however, is neither efficient nor practical for storing e.g. numerical floating-point values with high precision. Package-portability can only be attained through world-wide standards on how control structures and soft-coded control algorithms are to be stored on external files, a problem which is further discussed in Chapter 6.2.

Probably the most practical approach to the solution of this problem is to use a highly portable, but slow and rather limited, standardized format for communication *between* packages, and to adapt a commercially available data-bank to the requirements of CACSD to store data away for restricted re-use by the individual CACSD package itself. Such developments, however, are major undertakings worth deeper studies of their own.

3.13 Conclusions

In this chapter, we have discussed some important properties of interactive user interfaces to CACSD packages. Most of these properties were illustrated through examples from IMPACT, the CACSD package that, in our view, provides currently the most advanced user interface of all available CACSD packages.

Typical of all developments of interactive software, the design and implementation of the IMPACT user interface has been more of an *engineering task* than a scientific endeavor – it is the result of numerous compromises between factors such as expression power/ease-of-use, algorithmic completeness/package compactness, speed of execution/implementation time, etc. In retrospect, the following general rules have crystallized out as being imperative for any design of CACSD user-interfaces:

- *The basic commands of an interactive environment must be fast yet flexible.* When a program is used for the first time or when an advanced problem is to be solved, one expects to face difficulties (and will be positively surprised when this does not happen). However, when solving simple problems, the path one has to follow must be fast, easy and self-evident. If the user has to know just as much to solve simple problems as he needs for intricate ones, he (as well as the terminal) will get turned off! In this respect, the simplicity of the MATLAB "matrix interface" is ideal.

- *CACSD packages should support an algorithmic interface.* No package designer can foresee the details of every problem to be solved by his program. Therefore, no extensive fixed paths should be pre-programmed. Instead, compact commands should be made available to the user together with a mechanism to create his own paths (by means of interactively defined macros and/or functions). Moreover, no package can support all algorithms usable in control theory. Hence the user should be able to define new algorithms in a simple manner, preferably also as interactively defined functions. This requires a structured and powerful command-language interface.

- *The transition from basic to advanced use must be gradual.* The user of a CACSD tool is confronted with two different complexities: the complexity of the user interface and the complexity of the underlying theory/algorithms. In both cases, extra guidance is needed for novice users. In the package IMPACT, the user calling complex functions can switch over to the question-and-answer interface of the query-facility at will.

- *The system must be transparent.* The user should not only be in control of the executed actions. He should also be in control of the environment in which this execution takes place. This requires an interface with no hidden entities. Algorithms should be incorporated as stand-alone functions. All transfer of information must be made over user-specified parameters. In particular, separately callable functions must never share data over some hidden interface.

- *Small and large systems should be equally treated by the user-interface.* Many numerical control algorithms will work accurately only up to a certain system order. Ideally, these algorithms will warn the user whenever this limit is exceeded, and thereafter "gracefully deteriorate". This leaves the user "informed and in control". However, many traditionally implemented CACSD packages retain a fixed dimensional cut-off limit. This simplifies package implementation, but severely jeopardizes the usability of the package.

- *The system must be able to communicate with the external world.* As a minimum, it must be possible to store and enter data for later use and to communicate with other programs implementing any additional algorithms needed in the complete design. For flexible external data-handling, the employment of a special-purpose CACSD data-base is needed.

Chapter 4

DATA STRUCTURES FOR CACSD

4.1 Introduction

Three "buzz-phrases" have dominated the software world during the past two decades:

- **modular** overall **design**
- **structured** procedure-level **programming** and
- the use of descriptive **data structures**.

The software-principles associated with each of these three phrases are pertinent for the *implementation* of any CACSD package (see Chapter 5). Moreover, in Chapter 3 we stressed that these principles are just as relevant to the overall *design* of the *user-interface* of such a package. Thereby, we treated man-machine interface related issues associated with the first two of the mentioned three phrases. In this chapter, we will elaborate on the *data-structures* to be visible at a CACSD user interface. It will be shown that the data-structures supported by a CACSD package (or rather the absence thereof) delimits its versatility and extendability quite brusquely. Therefore, a set of data structures necessary and sufficient for most CACSD applications will be defined. As before, all examples not otherwise indicated are taken from IMPACT.

Many packages used in control theory perform all their operations on one single data structure, such as

- **real matrices**
  MATOPS (*Grepper et al., 1977*)
- **complex matrices**
  MATLAB (*Moler, 1980*)
- **polynomial structures**
  POLOPS (*Grepper et al., 1977*)
- **symbolic differential- and/or difference-equations**
  SIMNON (*Åström, 1982*)
- **general symbolic representations**
  MACSYMA (*1986*)

All of these single-structured programs are special-purpose packages. For example, SIMNON is intended for the simulation of continuous/sampled-data systems. Contrasting these packages, the so-called matrix-environments (supporting complex matrices only) are marketed as "general purpose control packages" (e.g. the MATLAB code-extensions CTRL-C (*Little et al., 1984*) and MATRIX$_X$ (*Walker et al., 1982*)). As long as it is our aim to analyze/synthesize linear systems in the time domain, the complex matrix is indeed an adequate data structure, since each system can be described by four such matrices. On the other hand, if we work in the frequency domain, we would like to describe our systems by transfer function matrices. This four-dimensional structure (cf. Section 4.4) cannot readily be represented by two-dimensional matrices. Therefore, the above mentioned MATLAB-extensions limit their frequency operations to the SISO-case, where a transfer function can still be represented as two one-dimensional vectors (containing the coefficients of the numerator and denominator polynomials, respectively). More intricate frequency-domain studies cannot be formulated in a straight-forward manner. This leads us to the following conclusions:

- *Software packages must support all descriptive units used by human specialists in the field.* The main reason why pocket calculators are not too useful in control theory is neither that they do not contain the right algorithms (today's calculators can be programmed) nor that they are too slow (which they are). It is the total lack of appropriate data structures that makes pocket calculators unsuitable for our task. Control engineers mainly work with matrices and/or polynomial structures (for time and frequency domain operations, respectively) as well as graphical representations originating from these kind of data structures. Such data structures are not available on pocket calculators. Unfortunately, they are not available in many CACSD packages either (!).

- *Data representations should not be simulated.* While the "matrix environments" support one data structure only, namely the complex matrix, and (ab)use this representation for all storage needs (as "matrix booleans", "matrix text strings", "matrix polynomials", "matrix time responses", etc), other approaches use strongly dedicated data structures for system representations, signals, frequency representations, etc. (Maciejowski, 1984).

  The latter approach is safer to use for the control engineer (for example, a matrix A denoting a continuous state-matrix is indicated as such, and no mixup with its corresponding sampled system equivalent can be made). On the other hand, such a dedicated approach is not as flexible when defining new algorithms. Program-code of a command-language supporting dedicated structures will contain a large number of conversion calls.

  It is the opinion of the author that a middle road is the optimal answer. Structurally different data (e.g. a matrix and a transfer function) should be stored in separate data structures, however, data with only semantical variation (e.g. the state-matrices of a continuous and a discrete system) should be treated equally by the system.

In the following, we will list the data-structures we deem necessary in a control environment. Also, we will discuss the applicability of overloaded arithmetic operations on these structures.

4.2 Matrices

The most basic, and in some cases the only (!), data structure of modern CACSD packages is the complex matrix. Vectors, scalars, and real matrices are normally treated as special cases with no separate data representation.

In IMPACT, as in MATLAB, all matrices are stored away using complex elements of high precision. Matrix input is made using the syntax presented already in Chapter 3. For example:

```
A = [1,2,3
     4,5,6
     7,8,9];
```

constructs a 3*3 matrix `A`. If the column vector `B` has been entered as

```
B = [1.5
     4.3
     1];
```

or

```
B = [1.5 ; 4.3 ; 1];
```

the equation $A * x = B$ can be solved by

```
X = INV(A)*B
```

(in which case the inverse of `A` is explicitly calculated and multiplied with the vector `B`) or

```
X = A \ B
```

(in which case `X` is calculated using Gaussian elimination without inverting `A`).

4.3 Polynomial matrices

A polynomial matrix is a matrix where each element is a polynomial with (complex) coefficients, such as

$$\begin{bmatrix} 2+3*p+1*p^2 & 3+1*p \\ 4+4*p+1*p^2 & 1+3*p+3*p^2+1*p^3 \end{bmatrix}$$

Polynomial matrices are entered into IMPACT using a short-form notation similar to that used for normal matrices. For example, the input line

```
Q = [ 2^3^1, 3^1 ; 4^4^1, 1^3^3^1 ]
```

will result in the above polynomial matrix. Although IMPACT stores away all numerical data in a complex high precision floating format, IMPACT also retains information on the "complexity" of each structure. Hence, the fact that the just entered polynomial matrix Q contained integer real-only elements only is recognized. IMPACT assigns each mathematical structure one of the values INTEGER_REAL_ONLY, INTEGER_COMPLEX, FLOAT_REAL_ONLY and FLOAT_COMPLEX, this information may be used by individual algorithms for checking the validity of certain operations. For example, in the expression A**x where A is a polynomial matrix, x must be an *integer* scalar. The complexity information is also employed for selecting the proper output format. Thus, Q is displayed on the screen as

```
Q(p)       =
   2.          3.
  +3.*p       +1.*p
  +1.*p**2

   4.          1.
  +4.*p       +3.*p
  +1.*p**2    +3.*p**2
              +1.*p**3
```

An alternative way of entering the polynomial matrix Q would be to use the system variable _P, which has been predefined as

```
P = [^1];
```

Thereafter the polynomial matrix Q can be entered as

```
Q = [2 + 3*_P + _P**2, 3 + _P
     4 + 4*_P + _P**2, 1 + 3*_P + 3+_P**2 + _P**3]
```

The predefined variable _P is one of several system-defined, read-only variables all having a name commencing with an underscore (*EAGLES/Controls, 1986*). Other system variable include _I for an imaginary value 1, _EPS for the system-dependent machine resolution, _E for the natural logarithmic base, and _PI. No names of user variables may *commence* with an underscore, however, the underscore is a valid character *within* user-defined names.

The basic matrix operations addition, subtraction, and multiplication may be used on polynomial matrices (using the symbols +, -, and *) if the basic dimensional rules are satisfied. For example, the input lines

```
Z     =  [1+_P  , 2*_P];
WROW  =  [1     , 2+2*_P];
WCOL  =  WROW';
XADD  =  Z + WROW  , XMULT = Z * WCOL
```

(where the ' operator denotes the conjugate complex transpose) will result in the output

```
XADD(p)     =
    2.          2.
   +1.*p       +4.*p

XMULT(p)    =
    1.
   +5.*p
   +4.*p**2
```

Until now, all polynomial matrices have been entered in a non-factorized manner, specified through all non-zero coefficients of the polynomial elements. Polynomials can also be represented by their factors, hence

$$\begin{bmatrix} (p+1)*(p+2) & (p+3) \\ (p+2)*(p+2) & (p+1)*(p+1)*(p+1) \end{bmatrix}$$

is the factorized version of the above introduced polynomial matrix. This factorized structure can be obtained in IMPACT though the command

```
QF  = FACTOR (Q)
```

which then displays the result

```
QF(p)    =
    (p + 1.)        (p + 3.)
   *(p + 2.)

    (p + 2.)        (p + 1.)
   *(p + 2.)       *(p + 1.)
                   *(p + 1.)
```

It is of course also possible to enter factorized polynomial matrices directly:

```
QF = [-1|-2,  |-3
       -2|-2, -1|-1|-1]
```

Due to the ill-conditioned re-factorization, operations on factored polynomial matrices, where at least part of the factors need to be defactorized before the operation (like addition), can be extremely badly conditioned. In IMPACT, the user is responsible for testing the accuracy of the result. However, IMPACT provides the user with a few tools to help him with the verification of results: a smaller computer word-length can be simulated by specifying the accuracy to be used in each arithmetic operation (as in MATLAB). This allows the testing of the error-propagation in the used algorithm. In addition, IMPACT will issue a warning message each time a detectable ill-conditioned operation is performed.

4.4 Transfer-function matrices

Only in special cases is the inverse of a polynomial matrix another polynomial matrix. However, the inverse of a polynomial matrix can (as long as the matrix is non-singular) *always* be defined as a matrix with rational function elements, a so-called transfer-function matrix.

Transfer-function matrices are entered in a manner similar to that used by polynomial matrix entry. The input sequence

```
G = [ 1/_P      , 1/(_P+1)
      1/(_P+1)  , 1/(_P*(_P+1)) ];

G = FACTOR(G)
```

and its short-form version

```
G = ONES(2) ./ [ |0, |-1; |-1, 0|-1 ]
```

(where `ONES(2)` returns a 2*2 matrix filled with ones and `./` denotes an element-by-element division) both result in the factored 2*2 transfer-function matrix

$$G(p) = \begin{bmatrix} \dfrac{1}{p} & \dfrac{1}{(p+1)} \\ \dfrac{1}{(p+1)} & \dfrac{1}{p*(p+1)} \end{bmatrix}$$

In control theory, transfer-function matrices are used to describe systems in the frequency domain. Interestingly enough, many mathematical operations on transfer-functions have physical meaning. For example, the multiplication in reverse order of two systems corresponds to a cascading and the addition of two systems corresponds to a parallel connection, as shown in Figure 4.1. Thus, the multiplication and addition operators have been **overloaded**[1] to work on transfer function matrices according to standard mathematical rules, giving IMPACT basic

[1]The term **overloading** derives from Ada, in IMPACT it describes the multiple use of one operator symbol or subprogram name to define several, separate algorithms operating on *different* data structures.

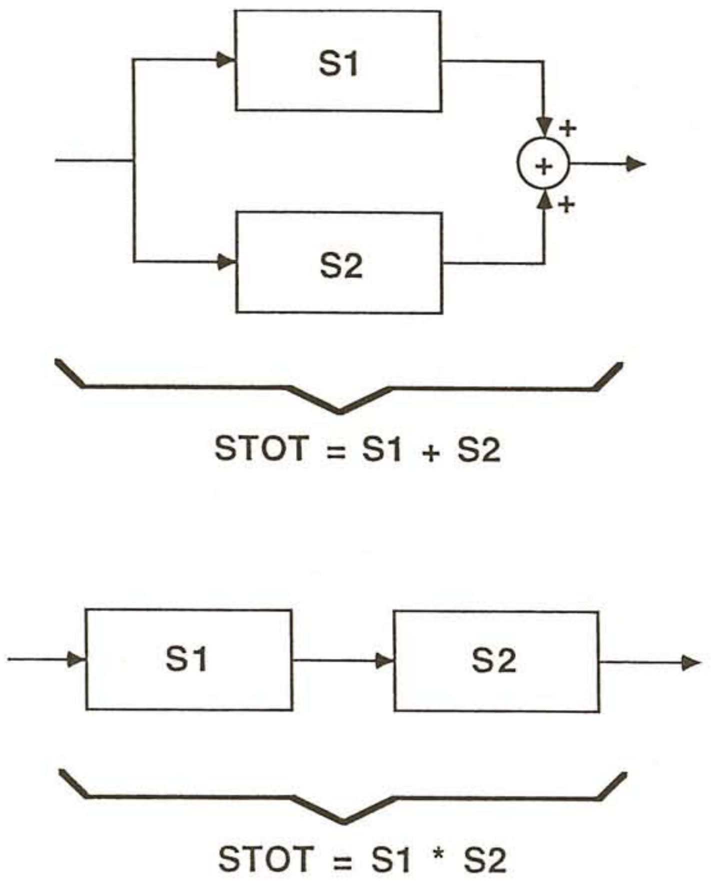


Figure 4.1: Series and parallel connection.

systems interconnection capabilities in the frequency domain for free. In the next section, we will see how the same operators are overloaded on systems described in the time domain.

A topological structure very common in control theory is the feedback loop as illustrated in Figure 4.2 The total transfer-function of this loop can either be described directly using the formula:

```
GTOT = G / (1 + G*H)
```

or through the use of the special feedback operator `\\` (which does not correspond to any trivial mathematical operation):

```
GTOT = G \\ (-H)
```

("`G` fed back with `-H`").

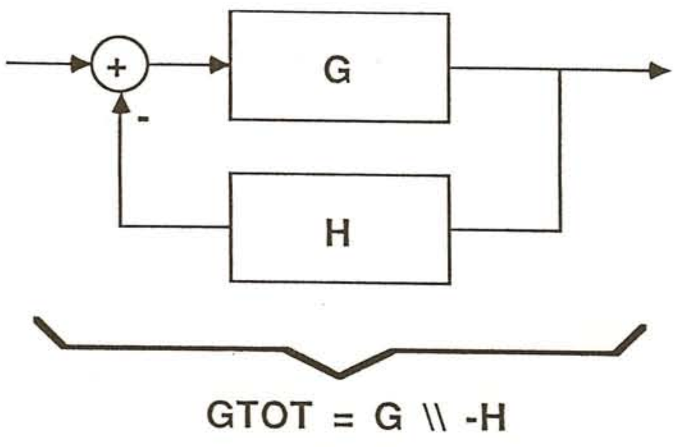


Figure 4.2: Feedback loop.

To illustrate the versatility of IMPACT in modeling control-related systems, the benchmark cable-spool system depicted in Figure 4.3 will be used. Linear approximations are used for the motor and tachometer, as shown in the block-diagram picture of Figure 4.3 In this section we assume that we can use a linear model for the spool, in Section 4.9 we treat the case with a nonlinear model of the spool.

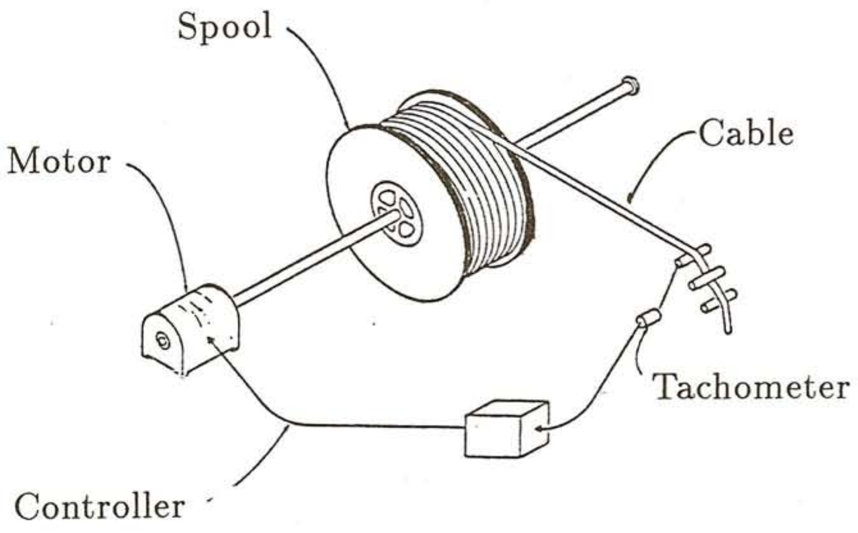
We will start by defining representations for all subsystems as shown in Figure 4.3. We commence by defining the proportional controller KP, and thereafter we define the three rational functions TACHO, MOTOR and SPOOL:

```
KP    = 1;

TACHO = 3/[1^0.5];

MOTOR = 7./[1^1];

SPOOL = 0.019/[^1];
```

Now, we can calculate the total transfer function of our system as

```
SYS = ( SPOOL * MOTOR * KP \\ - TACHO ) * 3;
```

where SYS of course is again a rational function scalar.



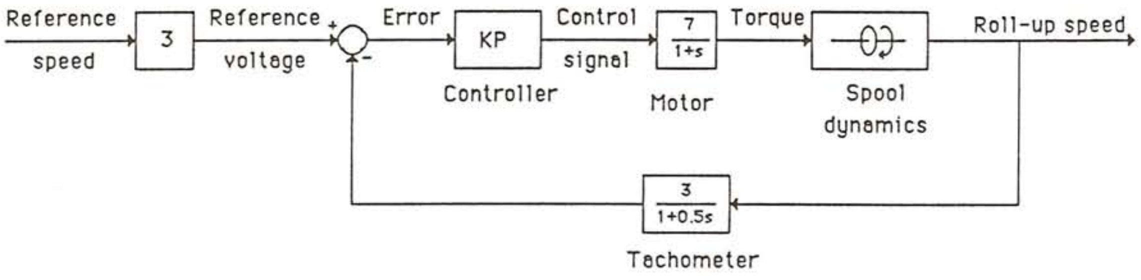


Figure 4.3: Cable roll system, physical setup and block diagram description

4.5 System descriptions

In the time domain, a linear system is normally described by four different matrices A, B, C, and D:

$$\begin{aligned} \dot{x} &= A * x + B * u \\ y &= C * x + D * u \end{aligned}$$

where x is the state vector, u is the input (vector) and y is the output (vector). As this is a very common representation, IMPACT provides the user with a special data-structure, the linear system description. Given three matrices A, B, and C of right dimensions, the function LINCONT will form a continuous linear system description out of these matrices,

```
CSYS1 = LINCONT(A,B,C)
```

whereas LINDISC will form a discrete linear system description with a sampling rate of DT:

```
DSYS1 = LINDISC(F,G,H,DT)
```

The D-matrix was here assumed to be a matrix of correct dimensions with zero elements. If the user wants to define a D-matrix, this can be entered through the use of default redefinition:

```
CSYS2 = LINCONT(A,B,C,D=>DD)
```

will include the matrix DD as the direct-path matrix.

The inclusion of a special data structure for linear systems simplifies all calls to algorithms working on whole systems, as the user then has to specify only one parameter rather than three to four separate ones. However, the generality of the component matrices is not lost. These can at any time be accessed as normal matrices using a dot notation, thus

```
CS_EIG = EIG(CSYS1.A)
```

will return the poles of a SISO system. Alternatively, the function POLES handles MIMO as well as SISO systems, and therefore requires a complete linear system as parameter:

```
CS_EIG = POLES(CSYS1)
```

A linear system given in the time domain by three (four) matrices can always be transformed into a frequency representation, and vice versa. The transformation from the time to the frequency domain is given explicitly through the formula

```
G1 = C * INV(_P*EYE(A) - A) * B + D
```

This valid IMPACT statement determines the transfer-function matrix inverse of $(s * I - A)$ before the multiplications are carried out. Alternatively, a predefined IMPACT function TRANS directly implements the transformation, and thereby avoids this time-consuming and possibly badly conditioned polynomial operation:

```
G2 = TRANS(LINCONT(A,B,C))
```

A transfer-function matrix determined in this way is not unique, as each transfer-function component might have reducible factors. The function REDUCE will cancel common factors of each matrix component (using the machine tolerance, or any other given tolerance, to determine if two factors are equal or not):

```
G = REDUCE(TRANS(LINCONT(A,B,C)))
```

As the transformation from the frequency to the time-domain is not unique, IMPACT will provide the user with a range of transformations resulting in linear system descriptions in different canonical forms, such as the Jordan form, etc.

Mathematical operations on system descriptions should be defined such that the physical meaning is the same as if the same operation were performed on transfer-function matrices. Thus, if a system of 2nd order has been defined through the matrices

```
A = [1, 1
     0, 1];

B = [0
     1];

C = [1, 0];

SIMPLE = LINCONT(A,B,C);
```

the operation

```
CASC = SIMPLE * SIMPLE
```

describes a cascading of two identical subsystems SIMPLE, and as a result we obtain a system of order 4 with component matrices

```
CASC.A = [1, 1, 0, 0
          0, 1, 0, 0
          0, 0, 1, 1
          1, 0, 0, 1]

CASC.B = [0
          1
          0
          0]

CASC.C = [0, 0, 1, 0]
```

Notice that the dimension of the system matrix has doubled, just as the order of the physical system has doubled.

Apart from concatenation, feedback, and parallel connections, a more general interconnection facility for linear systems should be supported. Thereby, the total state-space system description of any physically realizable interconnection topology, as illustrated in Figure 4.4 can be calculated using a simple and yet not widely known algorithm by DeCarlo and Saeks (*1981*). To describe such general topologies, a special representation is needed. In IMPACT, an interconnecting `SYSTEM` structure having a syntax similar to that of soft-coded subprograms is available. To

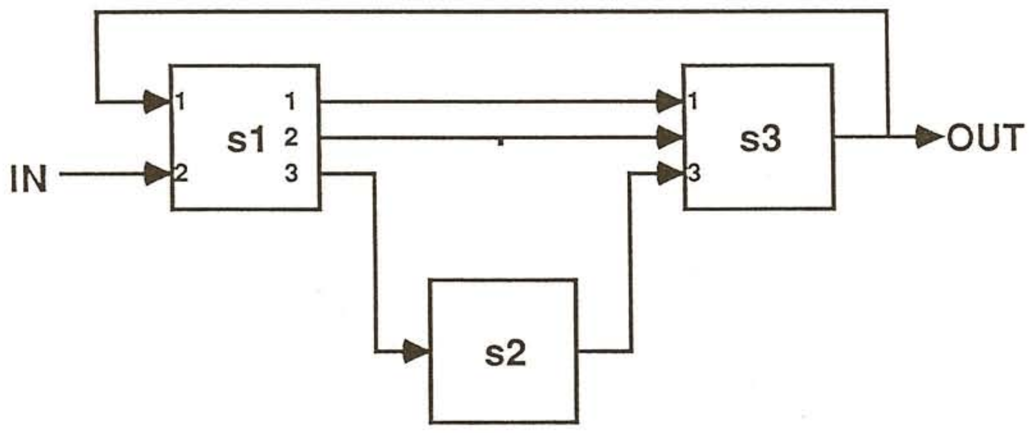


Figure 4.4: General interconnection topology.

describe the interconnections in Figure 4.4, an interconnection SYSTEM structure Sigma is created:

```
SYSTEM Sigma(s1,s2,s3) IS
CONNECT s1.IN(1)      = s3.OUT(1) = OUT,
        s1.IN(2)      = IN,
        s2.IN         = s1.OUT(3),
        s3.IN(1..2)   = s1.OUT(1..2),
        s3.IN(3)      = s2.OUT;
BEGIN
  NULL;
END Sigma;
```

In interconnection system definitions, the reserved words IN and OUT are used for the global input and output vectors, respectively. U and Y denote the input and output of the individual subsystems, respectively. The interconnection topology can thereafter be applied to any correctly dimensioned systems. For example, the interconnected systems SYS1 and SYS2 can be constructed as

```
SYS1 = SIGMA(S1,S2,S3);

SYS2 = SIGMA(S4,S5,S6);
```

if S1 .. S6 are defined as linear system descriptions having the correct number of inputs and outputs.

The systems SYS1 and SYS2 are linear systems only when the different component systems S1 .. S6 are linear. If any of these systems is nonlinear, the same kind of interconnection description may be used, but the used connecting algorithm will be symbolic rather than matrix-oriented, and the result is another nonlinear system description. This case is discussed in greater detail in Section 4.9.

4.6 Polynomial system descriptions

Transfer-function matrices and linear system descriptions are intended to represent linear systems in the frequency and time domains. A further common representation of linear systems is the polynomial-matrix representation which describes the system through a set of higher order differential equations (*Kailath, 1980*):

$$\begin{aligned} P(s) * z(t) &= Q(s) * u(t) \\ y(t) &= R(s) * z(t) + W(s) * u(t) \end{aligned}$$

Here, u is the input, y is the output, and z is the "partial state vector". Just as the "regular" linear system description, a polynomial-matrix system description contains four components (P, Q, R and W). Each of these components is a (factorized or non-factorized) polynomial matrix.

The method of entering and accessing polynomial-matrix representations is equivalent to what we have seen so far. For example, the system

$$\begin{bmatrix} 2s^2+3s+1 & -1 \\ -1 & s^2+4s+4 \end{bmatrix} * z = \begin{bmatrix} 1 \\ 0 \end{bmatrix} * u$$

$$y = \begin{bmatrix} 1 & 0 \\ s & 0 \end{bmatrix} * z$$

may be entered as

```
P = [1^3^2,-1;-1,4^4^1]
Q = [1;0]
R = [1,0;^1,0]
PSYS = LINPOLY(P,Q,R)
```

Algorithms for transforming linear systems within and between the three different representations transfer-function matrices, linear system descriptions and polynomial system descriptions exist (*Kailath, 1980*), and will be made available in IMPACT. However, when a transfer-function matrix representation with elements

$$G_{ij}(s) = \frac{P_{ij}(s)}{d(s)}$$

where $d(s)$ is the smallest common denominator of the transfer function matrix, and

$$ord(P_{ij}) > ord(d) \text{ for some pair } i, j$$

or when a polynomial system description with

$$ord(W) > 0$$

is transformed to a linear (A,B,C,D)-matrix system description, a polynomial D-matrix will result. To allow for these transformations as well, IMPACT will support polynomial D-matrices of linear system descriptions.

4.7 Domain and trajectory variables

A *domain* is a sequence of ordered, discrete values (domain points) which can be used to form the independent variable of a table. Generally, each of the domain points may take arbitrarily complex values, however, special domains exist. For example, a *time-domain* must have monotonously increasing real values. As an example of a time-domain,

```
TIME = LINDOM(0.,50.,0.1)
```

would define a sequence TIME with 501 elements, the first of which has the value 0 and the last the value 50, using an increment of 0.1. With the help of the '&'-operator, domains can be concatenated. For example would

```
PULSE_BASE = LINDOM(0,1,.01)&LINDOM(1.1,10,.1)&20
```

be a non-equidistant time-domain with 202 points.

A *trajectory* is a table of function values using a domain as independent variable. Such a table results from a variety of operations performed on domains. For instance would the operation

```
TRA = SIN(TIME)
```

result in a table where each entry contains an independent variable copied from the domain TIME and the sine-value thereof.

Mathematical operations are defined on trajectories using the same domain, e.g. would the operation

```
TRB = TRA + COS(TIME)
```

once again be a table with one row of values as function of the independent variable TIME, whereas

```
TRC = [TRA, COS(TIME), TRA + COS(TIME)];
```

would be a table where each entry is a row-vector with three elements. Note that TRB = TRC(3).

Furthermore, domains and trajectories can be used to *simulate* system behaviour. If we wish to simulate the step-response of our system in Figure 4.3 for 50 seconds with a 0.1 second resolution on the output, we can use the domain TIME to define the trajectory (signal)

```
U = ONES(TIME);
```

The simulation of the setup illustrated in Figure 4.5 is thereafter invoked as

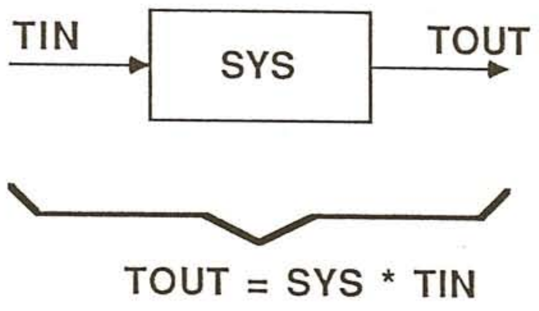


Figure 4.5: Simulation invoked thorough a multiplication between a system and a signal (trajectory).

```
VOUT = SYS * U
```

The results of this simulation will be stored in the trajectory VOUT sampled over the same independent range (domain) as TIME. This trajectory may then be plotted with default scaling by the command

```
PLOT(VOUT)
```

The use of the overloaded multiplication operator in this example is unambiguous and gives the user a very compact command for straight-forward simulation of linear and, as we will see in Section 4.9 on nonlinear systems. Moreover, this notation allows for a clear distinction between the simulated model and the simulation experiment, as proposed, among others, by Cellier (*1979*) and Zeigler (*1976, 1984*), and implemented in more recent simulation languages such as SYSMOD (*SYSMOD, 1986; Baker and Smart, 1983*):

$$result = model\_description * experiment\_descriptor$$

The *experiment_descriptor* is a normal trajectory describing the input driving function (if the system is autonomous, a domain is used). The domain points of this domain/trajectory are used as output communication points, and thereby also determine the domain point of the result trajectory.

For simple simulations, a trajectory over a predefined time-domain completely describes the experimental conditions for a simulation. In more involved cases, for example for simulation of stiff nonlinear systems, more experiment-related information such as integrational method, step-length, required accuracy, etc. is needed. This information must then be stored in "hidden" attributes of the trajectory. As these attributes have not yet been implemented in IMPACT, no detailed examples will be given. Nevertheless, the attributes will be accessible for examination and/or change to the user over a dot-notation as described in Section 4.8.3, or using a form-driven interface as discussed in Section 3.3.4.

Trajectories may be used not only for storing input and output signals for system simulations. Analogous to time domains, we define *frequency domains* as domains having monotonously increasing, normally logarithmically equidistant, imaginary values. For example would

```
FFRE = IMLOGDOM(1e-2,1e2,101);
```

create a domain with 101 logarithmically equidistant points between $0.01*i$ and $100*i$. Frequency domains may be used to calculate frequency responses of systems through the overloaded multiplication operator:

```
RESPONS = G1*FFRE;
```

RESPONS is now a trajectory with the real and imaginary value of the frequency response. This trajectory can now be plotted in any desired form through e.g.

```
PLOT(RESPONS,"BODE")
```

or

```
PLOT(RESPONS,"NYQUIST")
```

Alternatively to specifying the plot-kind (BODE, NYQUIST, etc.) at plot time, we could have assigned to the domain FFRE a hidden attribute of the same value:

```
FFRE = IMLOGDOM(1e-2,1e2,101,PLOT=>"BODE");
```

This value would then have been copied into the list of hidden attributes of the trajectory (RESPONS), and from there to the plot function.

If one wants to compare the Bode-diagrams of two transfer-functions, the trajectory vectors can be combined into larger trajectory vectors, and thereafter be plotted:

```
PL12 = [G1*FFRE,G2*FFRE];
PLOT(PL12)
```

On the plot, you will now find two different-colored/shaped curves from your two systems.

4.8 Non-numeric structures

Most CACSD packages are designed around a set of numerical algorithms. Even in packages supporting "graphical" methods (e.g. SUNS (*Atherton et al., 1985*) for limit cycle detection and CONCENTRIC (*Munro, 1979*) for multi-variable frequency-domain design), the underlying operations are of a numerical nature with numeric results being converted to graphical form for display. Hence, few CACSD packages directly support non-numeric structures. However, there are several instances where such non-numeric structures may prove useful in CACSD packages, for example in the form of

- Symbolic representations.
- Elementary programming structures.
- Composite structures with each element again being of a numeric or non-numeric nature.

In the following, we will discuss the applicability of each of the three cited classes of structures to CACSD packages. It should be noted that general symbolic representations, as described in the next sections, have not (yet) been implemented in IMPACT.

4.8.1 Symbolic representations in CACSD

None of the presently wide-spread CACSD programs allow for general symbolic manipulations. The overwhelming complexity of symbolic manipulation packages with implementation times of ten or more man-years together with a certain lack of expertise by the numerically oriented control engineers constructing CACSD packages have hitherto hampered the incorporation of any general symbolic processing power into control packages. Hence, control engineers wishing to perform symbolic or partially symbolic calculations have to resort to general purpose symbolic manipulation packages, such as MACSYMA (*1986*) or REDUCE (*Hearn, 1973*). However, although these packages are extremely powerful, they are not tuned for solving control problems. This forces each control-oriented users to construct his own customized operating environment before he can solve control problems with reasonable comfort.

Despite the present lack of control-oriented symbolic manipulation packages, (partial) symbolic processing has great potential in CACSD. For example, after specifying a linear model that contains a number of non-numeric parameters in the form

```
A = [1, 2
     0, 5*%M%+6];

B = ...
```

where %M% is a parameter with yet unspecified numerical value, results deriving from calculations on this model would contain factors of %M%. Thus the influence of certain physical/control parameters on the overall system description/behaviour can be studied. Note that it would not be very wise to specify the whole system in symbolic form, as the well known complexity-explosion of symbolic calculations would render quite uninterpretable results.

In the future, symbolic processing could be made available in CACSD packages either by embedding a general symbolic program into the CACSD package, or through an indirect link to an independent symbolic software package. However, already now some CACSD programs contain certain symbolic processing power, allowing for the modeling of nonlinear systems using an internal symbolic representation. These models are then either compiled/interpreted for nonlinear simulation, or a symbolic/numerical linearization algorithm transforms the models for further use in the linear parts of the package(s). In this manner, the powerful simulator ACSL has been incorporated as a nonlinear simulation environment within the CACSD package CTRL-C (*CTRL-C, 1986*).

In a later section, we will discuss the user-representation of symbolically stored nonlinear models.

4.8.2 Elementary non-numeric structures

Conventional, strongly typed computer languages such as Pascal, Modula and Ada offer the programmer several elementary non-numeric data-structures, including boolean, string, enumerated, and set types. Of these, boolean representations must be present in command-driven CACSD-packages for use within IF, WHILE constructions, etc. Strings are e.g. useful to pass textual information (titles etc.) to routines producing graphs or other reports. The support of enumerated and set types would enhance the general expression-power of the command-language, e.g. for the implementation of different menu-driven schemes.

Some packages, for example the "matrix-environments", allow matrix-structures to be *interpreted* as logical (BOOLEAN) structures. A possible scheme is to interpret all non-zero numerical values as TRUE and all zero values as FALSE. Assuming all values are non-negative, an addition then corresponds to an (expensive) OR-operation and an element-by-element multiplication is equal to an AND-operation (assuming we work on matrix structures). The main drawbacks of this approach are higher error-rates due to the weak typing, and poor readability of the resulting soft-coded programs.

In IMPACT, boolean variables may be created and manipulated using boolean expressions and the logical operations AND, OR and NOT. Boolean elements may be composed using the structures presented in the next section.

Analogous to the case of "overloaded" boolean matrices, the so-called matrix-environments allow vector structures to be interpreted as strings (for example by interpreting the individual elements as ASCII-code). This avoids the introduction of new types, however, it also means that "HELLO" + "$$100" will be displayed as "li|" (note that the displayed string is shortened as a result of a delete character!). In IMPACT, special string-variables of variant length can be created. These string variables may be manipulated in a meaningful way with the operations "+", "-", as well as extracted as substrings:

```
S  = "Time-response of system S1 with ";
S1 = "M1 = 5";
S2 = "M1 = 10";

T1 = S + S1;
T2 = S + S2;
T3 = "Frequency" + (T1 - "Time");
T4 = S(1..4);
```

will produce four strings T1..T4 with the content

```
"Time-response of system S1 with M1 = 5"
"Time-response of system S1 with M1 = 10"
"Frequency-response of system S1 with M1 = 5"
"Time"
```

4.8.3 Composite structures

In conventional programming languages, composite structures such as arrays and records enable illustrative and robust data abstractions. Such clustering of information is useful also in interactive CACSD packages to enhance the expression power of command-languages (as with the previously discussed 4 matrices that were combined by the LINCONT operator to form one linear system description), and to aid the user in managing and structuring his data.

Three approaches to clustering in interactive environments could be taken:

- *Predefined cluster structures*, such as all previously discussed numeric structures. While most CACSD packages limit these predefined clusters to basic mathematical structures such as matrices and trajectories, some authors suggest a much more rigid data organization with only one data structure: **the system** (*Maciejowski, 1984, 1985; Mason et al., 1985*). Such a system must then contain some or all of a range of predefined entities, such as nonlinear equations, a linearized system description, a frequency response of this linearized representation, etc. A slightly simplification of the rigid structure proposed by Maciejowski is shown in Figure 4.6. This approach structures the data in a manner natural to the control field, but also severely limits the options of the user.
- *Strongly but dynamically typed structures.* Similarly to the typing used in regular computer languages, the user may (dynamically) declare the elements and/or dimensions of a new record/array type, and thereafter create and use variables of this type. This approach is advantageous during semi-interactive definition of new soft-coded subprograms where typing would enhance program robustness. However, in a fully-interactive command-mode, such an approach clashes with the requirements of a transparent and flexible environment where the user dynamically creates new structures as he needs them.
- *Loosely typed structures* which are created "at the whim" of the user. Using this approach and allowing for user-defined records, arbitrary elements of variable dimensions could be added and deleted to/from existing records at any time.

As discussed in Chapter 3, the IMPACT command language has been designed considering the conflicting goals of algorithmic robustness and flexibility/ease-of-use. Consequently, a compromise has been sought between robust typing schemes and flexible but sometimes spuriously "free" clustering.

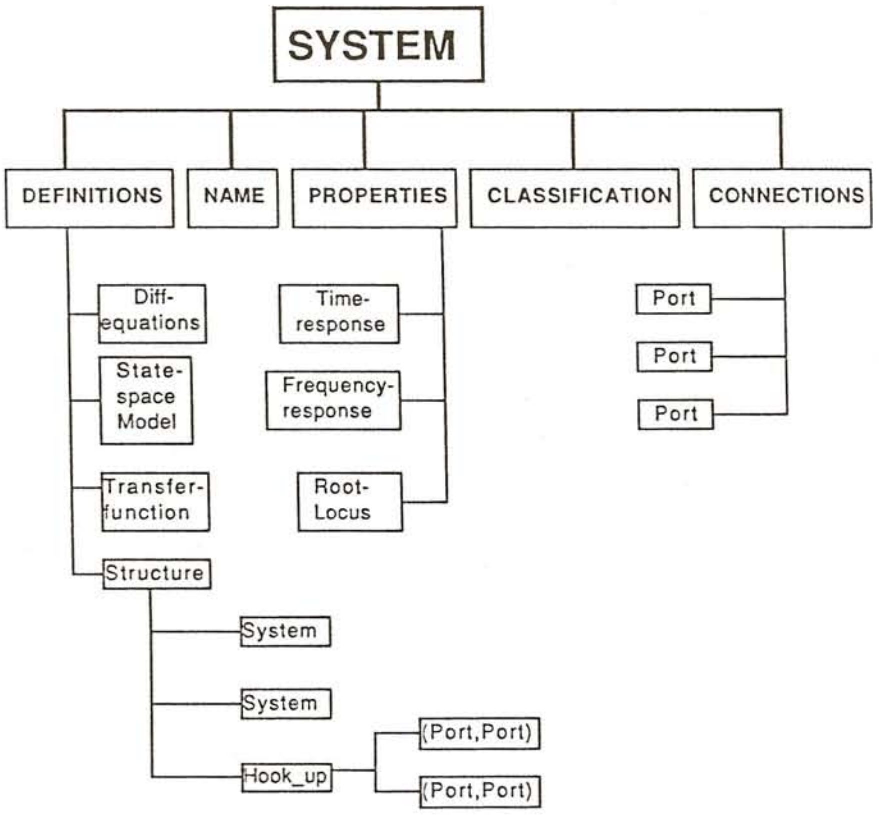


Figure 4.6: Predefined, rigid structure for a fictious CACSD package as suggested by Maciejowski. Some of the elements in this figure have further internal structures, hierarchical modelling is possible.

The typing in IMPACT is most rigid for basic numerical clusters (such as matrices, transfer-function matrices and trajectories) which are all of fixed types but with free dimensions. Only certain elements of predefined record types, such as linear system descriptions, have dimensional constraints to make the components compatible.

In subprogram definitions, a semi-strong typing is recommended. In IMPACT, type specification may be supplied for parameters and local variables of hard- and soft-coded subprograms, in which case type-checking is performed by each call. As opposed to conventional programming languages, this type-checking is "semi-strong", meaning that upwards compatible structures pass the test. For example, if a function to return the maximum order of a polynomial matrix has been defined using the type POLY_MATRIX (which is one of several predefined system types)

```
FUNCTION Order(p_poly : IN POLY_MATRIX)
                RETURN     INTEGER IS
BEGIN
  ...
```

all calls to this function where the parameter p_poly is a factorized polynomial matrix, a non-factorized polynomial matrix or a complex matrix are legal. The matrix is thereby treated as a special case of a polynomial matrix of degree 0.

Users may define new clustered types and use instantiations of these types as in strongly typed programming languages. This feature may for example be used to increase the robustness and readability of soft-coded subprograms, where user-defined declarations such as

```
FUNCTION Xyz(...) RETURN ... IS

  TYPE freq_sys IS RECORD
                     name  : STRING;
                     coeff : NF_TRANS_FUNC_MATRIX;
                     fact  : F_TRANS_FUNC_MATRIX;
                     bode  : TRAJECTORY;
                   END RECORD;

  open_loop, closed_loop : freq_sys;
BEGIN
  ...
```

are allowed. The robustness is thereby increased at the cost of a more complex subprogram definition and a slightly increased execution time due to the additional type checking. For example, if an incompatible assignment to a record-element is performed within the body, an exception will be raised, and the user will be supplied with the appropriate error message. Assuming the elements of the record had not been typed, the error would have been detected later, for example at the first usage of the erroneous element.

To enable type-checking of parameters of user-defined structured types, types may be "inherited" according to normal scope rules. However, as any type declarations normally is of a more permanent basis than dynamically created variables, a type declarations should be protected against overwrite and deletion Thus, all types interactively entered, such as

```
TYPE freq_sys IS RECORD
                   name  : STRING;
                   coeff : NF_TRANS_FUNC_MATRIX;
                   fact  : F_TRANS_FUNC_MATRIX;
                   bode  : TRAJECTORY;
                 END RECORD;
```

must be automatically protected against any changes. The type can be explicitly unprotected through a call to subprogram `UNPROTECT(freq_sys)`, thereafter it may be deleted or modified (which possibly makes any user-defined subprograms relying on this type declaration useless or even erroneous). A companion procedure `PROTECT` should be available for protecting precious variables (etc.) from being destroyed by unindended overriding.

In interactive command-driven environments, strong typing of dynamically created variables should never be mandatory. IMPACT therefore allows, but never forces, the user to interactively define new types in addition to the predefined numerical ones. Moreover, the user wishing to group his data in an ad hoc manner may create fully dynamic records. Let us assume that a user wishes to dynamically create a record similar (but not identical) to the previously declared freq_syst. If the open-loop non-factorized transfer-function already is stored in the variable SYS1, the following self-explanatory operations would be legal:

```
freq1.name  = "SYSTEM 1";
freq1.coeff = sys1;
freq1.fact  = FACTOR(freq1.coeff);
freq1.xx    = "You may create new elements";
```

Note that a new variable `freq1` with one element (`name`) is automatically created during the first assignment. Thereafter, additional elements are added to this record. Records may of course be nested, such as in

```
freq1.yy.y1 = "A nested element"
freq1.yy.y2 = "Another nested element"
```

In addition to user-defined records where each data element is accessed with a dot-notation, data, in IMPACT, may be structured into so-called *indexed clusters*. Such clusters are flexible versions of the arrays in conventional programming languages in that a unitary typing of the array elements is not mandatory. Moreover, there is neither a limit on the number nor on the range of indices; indices may even be used discontinuously. To distinguish between the indices of predefined numerical structures and user-indexed clusters, square brackets are to delimit the cluster indices. Thus, the following commands would be legal (but not necessary meaningful) in IMPACT:

```
   A = [1,2;3,4];
   AA[1] = A;
   AA[2] = A + EYE;
   AA[5] = 5*EYE(A);
   AA[5](1,2) = AA[1](1,1);
```

Naturally, if these commands were followed by

```
   B = AA[3]
```

an error would occur, as `AA[3]` was never defined. Note the difference between

```
  A(I,I)
```

which denotes the complex element in row `I` and column `I` of the matrix `A`,

```
   A[I]
```

which denotes the I'th component of the one-dimensional indexed cluster A (where, of course, I has to be a defined, integer scalar variable), and

```
A.I
```

which denotes the component I of a user record A.

User-indexed structures are intended to simplify the description of repetitive but inhomogeneous actions. For example, if we wish to iteratively call a function user_design and store each of the intermediate results, the code

```
-- SS1 is assumed to be an already defined system.
k[0] = [1,2];
FOR i IN 1 .. n LOOP
  k[i] = User_design(ss1,k[i]);
END LOOP;
```

would give the final iterative result in `k[10]` and each intermediate results in `k[1]` through `k[9]`.

4.9 Nonlinear systems

Until now, we have discussed data structures suitable for representing linear systems in the time as well as in the frequency domain. Unfortunately, these structures do not always suffice. As we live in an imperfect world, control engineers often have to use nonlinear models consisting of differential and/or difference equations to describe real systems. Such models could be formulated using for example graphical definition packages such as MATRIX$_X$/SYSTEM_BUILD (*Shah et al., 1985*) or Hibliz (*Elmqvist and Mattsson, 1986*), conventional simulation languages following the CSSL'67 standard (*Augustin et al., 1967*) such as ACSL (*1986*), CSSL-IV (*1984*) or more modern, modular simulation languages SYSMOD (*SYSMOD, 1986; Baker and Smart, 1983*). However, only SYSTEM_BUILD integrated with MATRIX$_X$ can be classified as a CACSD package, the others can "only" be used for simulation and/or documentation (except for ACSL, which can be used together with the CACSD package CTRL-C (*CTRL-C, 1986*)). In control environments, we are not satisfied with simulation capabilities alone. Instead we expect to find a range of nonlinear tools such as

- a *nonlinear time-simulator.* It should be possible to describe nonlinear models separately from the simulation experiment to be performed. The experiment description facility should contain mechanisms to specify model-external driving functions which in themselves may be results of previous operations, e.g. previous simulations. The results of the simulation should be stored away in a manner such that they can be reused for other purposes by any parts of the control package.

- algorithms for *nonlinear controller design.*

- *nonlinear sensitivity analysis* either through linearization algorithms or by means of a worst-case multiple-simulation approach, a so-called range analysis (*Cellier, 1986*).

- a *linearizer* creating models for further treatment within the linear part of the control package. If a linear controller-design is made, the results of this design must be transported back to the nonlinear model for verificational simulations.

- tools for *hierarchical modeling* of larger nonlinear systems using subsystems in a modular fashion.

Not only does each of these facilities require separate sets of commands and underlying algorithms. Also, the user-defined models are accessed in completely different ways with the consequence that alternative connections/parameter-sets for each model may be needed. In the most general case, all of the following model interfaces may be needed for a generic model definition:

- *Physical parameters.* During the modeling and/or design phase, individual parameters of a physical system may not be known or may not yet be determined. In particular, the definition of predefined submodels with free parameters should be supported. Therefore, it should be possible to define model *types* (templates), from which instantiations with fixed parameter-values can be created.

- *Input and output signals.* In control environments where simulations are invoked as simple operations (e.g. as multiplication between a system and a time trajectory), inputs and outputs must be definable for each (sub-)system.

- *Initial conditions* and *equilibrium points.* Before a simulation can be started, the initial condition of each state variable must be specified. Similarly, equilibrium points for the state variables are needed by the linearizer.

- *submodel connections.* For simple submodel interconnections, the overloaded operations for parallel, series, and feedback connections suffice. However, for a general modular modeling, possible interconnections ("cuts") between submodels must be defined in each subsystem. These cuts are then connected to each other in the hierarchically higher system.

Hence, our descriptive language for nonlinear models must be a quite versatile description syntax supporting all these interfaces. In IMPACT, an attempt to support all these interfaces with a minimum of overhead has resulted in the definition of the nonlinear SYSTEM descriptions.

The nonlinear system descriptions of IMPACT must not be seen as just another nonlinear modeling language, but far more as an integration of a nonlinear modeling and simulation environment into a command-driven control environment. This allows the user to work with a unified user-interface during the entire design cycle. It also allows for a complete mixture between nonlinear (symbolic) and linear (numeric) models, something particularly useful when linear controllers are designed for nonlinear systems. Moreover, the algorithmic interface of IMPACT can be used as a flexible simulation environment for invoking complex simulation operations.

4.9.1 Nonlinear system descriptions

In this section, we will show how nonlinear operations can be invoked without increased complexity compared to the linear case. Returning to the example from Figure 4.3, the nonlinear model for the spool is assumed to be:

$$\begin{aligned} v &= r * av \\ \frac{dav}{dt} &= \frac{torque}{inertia} \\ inertia &= cable * r^4 + roll \\ \frac{dr}{dt} &= -k1 * av \\ k1 &= \frac{d * d}{2 * \pi * w} \end{aligned}$$

with the constant values

$$\begin{aligned} d &= 0.03 \\ w &= 0.6 \\ cable &= 23.5 \\ roll &= 2.1 \end{aligned}$$

where v is the roll-on/-off speed, r is the time-dependent radius of the roll, and av is the angular velocity of the spool. The two constants d and w denote the cable diameter and spool width, respectively. Typical values of w, d and the inertia constants *cable* and *roll* are indicated to the right.

As the overall model consists of more than a dozen lines, the risk of making errors during a line-by-line direct entry is relatively large. We therefore invoke the systems editor from within IMPACT and define the system using this editor as shown in Figure 4.7 This system definition declares a spool-template with not yet determined initial condition `r0` and system constants `d` and `w`. These free parameters of the defined model *must* be specified by all operations on the spool. The system-declaration header also includes information on input and output parameters to the system, which are implicitly used when we, for example, perform a simulation using commands identical to those used for linear systems:

```
SYSTEM Spool(r0,d,w     : SCALAR)
             IN torque : SCALAR
             RETURN v  : SCALAR IS
  --
  r       : STATE := r0;
  av      : STATE := 0.0;
  inertia : SCALAR;
  cable   : SCALAR := 23.5;
  roll    : SCALAR := 2.1;
  k1      : SCALAR := d*d/(2*_pi*w);
BEGIN
  v       = r*av;
  inertia = cable*r**4 - roll;
  av‘     = torque/inertia;
  r‘      = -k1*av;
END Spool;
```

Figure 4.7: Nonlinear IMPACT system. The system describes the nonlinear model of the spool of the cable-roll example.

```
VOUT=(SPOOL(1.2,0.03,0.6)*MOTOR*KP\\-TACHO)*3*U
```

In the shown examples, default integration parameters have been used during simulation. As described in Section 4.7, for complete simulation control, trajectories may contain additional information on integrational methods, step sizes, error conditions, et cetera.

If we wish to "freeze" the parameter-values and initial conditions, we may do so by creating a new nonlinear system as:

```
MYSPOOL = SPOOL(1.5,0.03,0.6);
```

Whenever a model is invoked with an incorrect number of parameters, and specially if formatted textual information has been included in the model definition, the powerful IMPACT query facility described in Section 3.10 will jump into action. For example, if we perform a simulation without specifying the free parameters, such as in

```
VOUT = ( SPOOL * MOTOR * KP \\ - TACHO )*3*U
```

IMPACT will prompt the user for missing parameters (user input is underlined:

```
S>>The nonlinear system SPOOL has been invoked
S>>with missing parameters. This system models
S>>the roll-off of a cable from a spool.
S>>Please enter missing parameters.
S>>
S>>Initial roll-diameter RO  (NO DEFAULT): 1.2
S>>Diameter of the cable D   (NO DEFAULT): 0.03
S>>Width of the spool    W   (NO DEFAULT): 0.6
```

If the user is uncertain about the meaning of a particular parameter, he can enter a HELP for further information.

```
S>>Initial roll-diameter RO  (NO DEFAULT): HELP
S>>This parameter indicates the initial diameter
S>>of the cable-spool (including the cable) at the
S>>outset of the simulation (this thickness is
S>>to be calculated as the radius of the spool
S>>plus the thickness of the cable-layers).
S>>
S>>Initial roll-diameter RO  (NO DEFAULT): 1.2
```

The system query facility is particularly useful for systems with many parameters, and for cases where the user and the constructor of a system are different people.

Other operations that can be performed on nonlinear systems are, with the exception of the hierarchical modeling, invoked through commands implemented in the regular command-language of IMPACT. For example, a linear state-space representation of SPOOL may be generated by symbolic linearization (*Rall, 1981; Joss, 1976*):

```
LINSPOOL = LINEARIZE(SPOOL(1.2,0.6,0.03));
```

Also here, the query facility would jump into action when either the function LINEARIZE or system SPOOL (or both) were specified with missing parameters. If LINEARIZE was specified without parameters, it would first ask for the system to linearize (SPOOL), and thereafter ask for any parameters of the model SPOOL (the query handler of LINEARIZE is programmed to allow for such a query within a subprogram call, cf. Section 3.10).

Yet another example illustrates the use of the IMPACT command language as an experimental frame for simulation runs. Assume that the parameters r0, d, and w are known with 10% accuracy, and that we wish to plot the envelope of the step-responses from all worst-cases of this parameter-variation (*Cellier, 1986*). We would then create a matrix where each row corresponds to one set of parameters:

```
MEAN = [1.2, 0.03, 0.6];
ERR  = [0.1, 0.1,  0.1];
RUNS = PERMUTE(MEAN,ERR,"RELATIVE")
```

resulting in

```
 RUNS =
   1.0800      0.0270       0.5400
   1.0800      0.0270       0.6600
   1.0800      0.0330       0.5400
   1.0800      0.0330       0.6600
   1.3200      0.0270       0.5400
   1.3200      0.0270       0.6600
   1.3200      0.0330       0.5400
   1.3200      0.0330       0.6600
```

A trajectory with the results from eight simulation runs using the above parameter sets is constructed through the commands

```
FOR INDEX IN RUNS(I,:) LOOP
 S(I)=(SPOOL(RUNS(I,1..3))*MOTOR*KP\\-TACHO)*3*U;
END LOOP;
```

The envelope of these eight time-responses is then obtained through the command

```
 PLOT(S,"ENVELOPE");
```

4.9.2 Modular system interconnections

For general interconnections not describable through the arithmetic operations and the feedback operator, the more general interconnection facility described in Section 4.5, may be used on nonlinear systems as well. Thereby, the valid specifications of the input and output connections of nonlinear systems are extended to be positional or named. Thus, the interconnection

```
SYS = ( SPOOL * MOTOR * KP \\ - TACHO ) *3
```

of our cable-roll example could also have been obtained through

```
SYSTEM Totsys(spool,motor,kp,tacho)
              IN u
              RETURN y IS
CONNECT
  kp.IN         = IN - spool.v,
  motor.IN      = kp.OUT,
  spool.torque  = motor.OUT,
  y             = spool.v;
BEGIN
  NULL;
END Totsys;
```

Naturally, the result is again another nonlinear system.

Despite the general appearance of these interconnection tools, for true modularity in modeling we need to introduce yet another concept. While linear systems in transfer-function or state-space form have well defined inputs and outputs, this must not be the case for systems described in algebraic and/or differential equation form. Even a model of the simplest of systems, a resistor, may be used in two ways, depending on its surrounding connections, see Figure 4.8. Moreover, it is often not clear at the time of subsystem modelling exactly how the subsystem is to be connected to its surroundings (and thereby e.g. which of the two resistor models is needed at "connect-time"). Hence, for a modular design of large systems in IMPACT, the interface concepts of Elmqvist will be used. In his pioneer work, Elmqvist (*1978*)

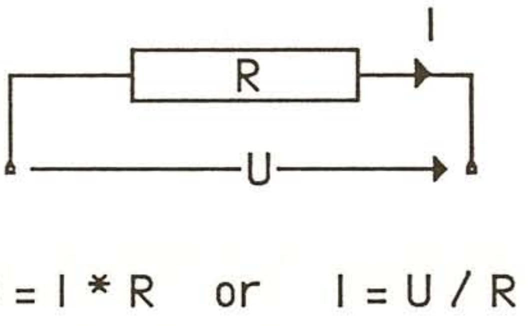


Figure 4.8: The two possible resistor models.

allows the system equations to be entered in any form, for example would Elmqvists program DYMOLA accept the equation set

```
torque  = av'*inertia;
av*r    = v;
inertia = cable*r**4 - roll;
r'      = -k1*av;
```

just as well as our original equations. I.e., statements are no longer assignment statements in a conventional form, but follow the more general syntax

$$expression = expression$$

A symbolic formulae manipulation program not only sorts equations "vertically" into executable order (as this is done in most CSSL's), but simultaneously sorts equations "horizontally" for the appropriate output variable. The sorter also checks the equations for completeness and consistency, and detects any algebraic loops requiring special treatment. This mechanism allows the user to enter equations as they were first obtained e.g. from physical laws.

To fully utilize the freedom of vertical and horizontal sorting over different submodels, it must be possible to define modules without having to specify whether a connection variable is an input or output signal. Elmqvist has shown that there generally exist two kinds of connecting variables; "ACROSS" variable which are set equal at the interfaces (as the voltages at the connection of three resistors) and "THROUGH" variables which are summed to zero at the interface (as the currents at the same connection).

These concepts of Elmquist will be incorporated in IMPACT as well. Returning to our example, if we wish to include the inertia of the motor in our system, the motor would have to be modelled in greater detail, and our spool model could be defined having the torque as a through variable, and the angular velocity as an across variable at the interface to the motor:

```
SYSTEM Spool(r0,d,w     : SCALAR)
             IN torque  : SCALAR
             RETURN v   : SCALAR IS
  CUT axis(torque  : THROUGH;
           av      : ACROSS);
      cable(v      : ACROSS);
  --
```

Note that this system has in/return parameters as well as cut declarations. This allows us to use the same model *either* as previously with fixed inputs/outputs *or* in a hierarchical model as follows: assuming the motor has a similar cut axis, then the two systems can be combined in a hierarchical system having the header

```
SYSTEM Motor_spool(spool, motor  : SYSTEM)
                   IN     uin    : SCALAR
                   RETURN vout   : SCALAR IS
  CONNECT spool.axis = motor.axis;
  CONNECT vout = spool.cable;
  CONNECT uin  = motor.uin
BEGIN
  NULL;
END Motor_spool;
```

This model can then again be used to form the system of Figure 4.3.

4.10 Conclusions

Recent trends in computing have led to the introduction of modern hardware (workstations) and software (interactive environments) where professionals of different technical fields work with conceptional entities of their speciality rather than underlying programming elements such as ARRAYS and LISTS. These entities usually correspond to the elements used by the same professionals when they work(ed) with pencil-and-paper or "real models". For example, the mechanical engineer works with geometric entities of his CAD-package, the numerical analyst is happy with the different numerical structures of matrix-environments such as MATLAB, and even a mathematician sometimes finds consolation in the intricacies of packages for symbolic manipulations.

Following this general trend, control engineers should also be supplied with the conceptual structures they are used to work with. This means that an ideal control environment should support entities for:

- numerical descriptions of systems (matrices, transfer functions, etc.),
- symbolic elements for general system-equations,
- graphical elements for the definition of system topologies,
- support of large-scale data management, e.g. in form of relational data-base support (cf. Section 3.12),
- support of small-scale data management, e.g. in form of spreadsheets, and
- graphical displays of numerical computations, possibly together with graphical interactivity for requirement specifications, etc.

The most popular present-day control-environments such as PC-MATLAB (*Moler et al., 1985*), CTRL-C (*CTRL-C, 1986*), and MATRIX$_X$ and SYSTEM_BUILD (*Shah et al., 1985*) only partially support the first and last groups and ignore symbolic and graphical system descriptions (MATRIX$_X$ together with SYSTEM_BUILD allows for a graphical input of topologies, a prototype system called MODEL-C to be used as companion to CTRL-C has been shown recently). Not a single system known to the author supports all six groups. Moreover, only few of the systems are truly versatile in their offerings of numerical structures, the most fundamental of the groups. In this chapter, we have shown that a large number of different numerical structures are needed to cover all methodologies used in control engineering, and that a "simulation" of some of these structures through other, simpler structures proves contra-productive due to the limitations then put on the systems.

Chapter 5

IMPLEMENTATION CONSIDERATIONS

5.1 Introduction

During implementation of the very first generation of interactive CACSD packages in the 1970's, software engineering principles and modern programming methodologies were seldom consciously used. Although it would be only too easy to use our enlighted eyes of the 80's to patronizingly condemn some of these programs as "unreliable-by-design", we should keep in mind that

- the packages of the 70's typically consisted of 80% numerical routines, 10% "glue" between these routines, and 10% user interfaces (*Agathoklis, 1986*). Most of the algorithms were individually developed, separately from the CACSD package (maybe even by different people), and inserted as stand-alone black-boxes. This kind of program development raised little need for any ingenious overall software design.
- FORTRAN was the "lingua franca" among scientific programmers. This ensured a fair portability of programs, but also delayed the introduction of modern programming methodologies such as abstract data structures and structured programming.
- Most CACSD packages were not large enough, or did not involve enough programmers, to induce the software man-

agement problems tormenting today's CACSD software developers. This was mostly due to the memory limitations of older computers (including mainframes), compelling the programmers to keep the individual programs small or to tediously specify overlays. Today, these memory problems are all but gone (virtual memory), but instead, software engineering methods (*Sommerville, 1985*) are becoming indispensable; ten years ago, this computer-science term had not even been coined. It is interesting to notice that even a powerful package like MATLAB (*Moler, 1980*) consisted of not more than some 7000 lines of (FORTRAN) source code, whereas modern control environments are easily more than one magnitude larger in size.

Although advances in computer science during the past two decades have influenced application fields such as CACSD to a certain extent, tradition and ignorance still foster outdated principles. Some programmers of today have themselves had formal education in structured programming and abstract data structures, but most of them were educated in FORTRAN, and like ones mother tongue, the first computer language mastered seems to be the one that most programmers feel particularly comfortable with. For these reasons and others, the vast majority of todays programmers still uses the most unstructured programming language available — FORTRAN. This tremendous gap between computer science theory and practice can only partially be bridged by the use of schemes for structured FORTRAN programming. Moreover, although standard FORTRAN code is quite portable, both FORTRAN'66 and FORTRAN'77 lack elements for system-calls and more intricate input/output — forcing the programmer to include system-dependencies into his code. As a consequence to all this, recent implementors of CACSD packages have considered using other programming languages, for example one of the following more widely spread languages:

- **Algol**. The first version of this language, Algol'60, made structured programming (almost) mandatory. A second version, Algol'68, was regarded as too complex and cumbersome to use, and for that reason never became very popular. Algol'60 remained popular in academic circles until the mid 70's, when it was superseded by Pascal. Today, Algol itself has survived in Eastern Europe only, whereas one of its more direct descendents (SIMULA'67, *Birtwistle et al., 1973*) still enjoys some popularity in particular in Scandinavia.

- **Pascal**. This language was designed by Wirth and Jensen (*1975*). It introduced a rich selection of data types and a consistent instruction set for structured programming, both prerequisites for reliable (CACSD) software. Because of its relative simplicity, Pascal has become one of the most popular programming languages in programming education, for the implementation of small to medium sized stand-alone programs, e.g. modestly sized parsers and compilers (*Bongulielmi, 1984*), and for simulation package translators (preprocessors) (*Baker, 1983*). For implementing large (CACSD) programs, the lack of standard language elements for a modular overall design (no modules, no separate compilation) and a rather weak input-output facility put Pascal at a disadvantage against languages such as Modula-2 and Ada.

- **Modula-2**. Wirth's (*1985*) most recent successor to Pascal has a syntax similar to that of Pascal, but includes language elements for modular design, flexible input-output, real-time programming, and foreign language access. Far better suited for implementing large programs than the previous languages, it still remains to be seen how widely-spread the use of Modula-2 will become. Modula-2 is presently gaining popularity among academics with good compilers running even on fairly small machines (e.g. MacIntosh and IBM-PC). Modula-2 lacks certain features described later in this chapter, for example discriminants and exception-handling.

- **Ada**. The reference manual of this language (*ANSI, 1983*) was developed according to specifications issued by the U.S. Department of Defense, and is now a required implementation language for embedded programs in many defense systems. Despite its background, Ada is not a "weapons oriented" product, but a general purpose programming language to be placed last in the chain Algol — Pascal — Modula-2 — Ada. The language has evolved from Pascal rather than Modula-2, yet it incorporates practically all new features of Modula-2, and a few additional ones of its own.

Although slow in coming, there are now some 25 vendors offering 55 validated Ada compilers (one validation per host-target computer configuration is required) (*AdaIC, 1986*), and more are expected. As for Modula-2, it still remains to be seen how well Ada will become accepted. However, already present commitments to Ada guarantee that a rather large set of utility modules (numerical algorithms, data-base programs, graphical drivers, etc.) will be made available in the future.

Ada has been criticized by many for being too large and too complex to be used efficiently. However, the language itself is modularly constructed making it convenient to use only a consistent subset of the language. This makes the complexity comparable to that of Modula-2. It is our experience that students with working knowledge of Pascal or Modula-2 need less than 2 weeks before they can use the rather large subset of Ada used in IMPACT.

- C. Among the hitherto mentioned structured programming languages, C (*Kernighan and Ritchie, 1978*) is somewhat of an outsider. It supports language elements for typing and structured programming, but does not enforce their use. The C language therefore opens up for efficient "trick programming", something generally appreciated by the people for which the language was originally created – systems programmers. Because of its close link to the UNIX operating system (UNIX is implemented in C), the future popularity of C will at least partially depend on the success story of UNIX. C is particularly powerful for the implementation of hardware-close programs, that is: hardware drivers, and system operating software in general where the use of C often results in more efficient run-time code. In the USA, C has meanwhile bypassed Pascal in popularity, and has become a de facto industry standard. At least one CACSD-package (the matrix environment PC-MATLAB; *Moler, 1985*) has meanwhile been implemented in C with very good success.

- **PROLOG** (*Clocksin and Mellish, 1984*) and **LISP** (*Winston and Horn, 1981*). These non-procedural languages might be considered for CACSD projects where control design algorithms are combined with expert systems, other artificial intelligence elements, or more general symbolic processing elements. Hitherto, these languages have only been used in CACSD to implement semi-independent "intelligent" units connected to parts implemented in normal procedural languages for the numerical, graphical and/or user communication tasks (*Trankle, 1986; Larsson and Persson, 1986*).

As previously stated, most CACSD-packages and CACSD-libraries have until now been implemented in FORTRAN. With the development of IMPACT, one of the first decisions to be taken was NOT to use FORTRAN. Apart from the well-known drawbacks of this language as discussed previously, we felt that it is the obligation of academic research groups to be in the forefront of their respective fields, and that generally valuable experiences for the field of CACSD could be gained by investigating alternative implementation languages. Hence, several FORTRAN-alternatives were evaluated, and Ada was made the language of our choice. Not directly depending on the revenue of our efforts, we were able to take a somewhat more risky decision, a decision which might not (yet) have been justified in the commercial world.

In the remainder of this chapter, we will discuss different implementational aspects of larger CACSD packages. Each aspect will be illustrated by appropriate extracts from IMPACT together with comments on the suitability of Ada as an implementation language.

Some of the IMPACT code-extractions of this chapter have been slightly edited/simplified for clearer illustrations. Such discrepancies with the actual IMPACT code will not be specially marked.

5.2 Programming conventions

The development of large software packages, such as modern integrated CACSD packages, is a major engineering task involving a large number of persons over several years. Left alone, each of these persons would have his own approaches to software design and programming style, resulting in a collection of heterogeneous modules. The discrepancies between these modules could range from deviating naming-conventions over varying input-output formats to different data structures. Such inconsistencies would not only decrease the readability of the total program, but also increase the risk of module incompatibilities. Therefore, a certain style of program design (coding rules) has to be adapted for larger software projects to ensure readability, reliability, and maintainability of the code.

As mentioned previously, the portability and re-usability of Ada programs was one of the primary goals behind the design of that language. This is not only reflected in the strict requirements on compatibility between Ada compilers and run-time systems, which have to be *validated* before they may be released, but also in the plans for official style-rules on how Ada programs "are supposed to look like". The first draft of these style-rules (*Roski, 1986*) is too general for individual projects as it mostly limits the use of certain language elements to increase programming security (also see *VanNeste, 1986*). Therefore, each software group/company must specify additional rules related to the particular requirements. Thereby, it is important that these rules are formulated in such a way that the fantasy and free inspiration of the programmer is not restricted. For IMPACT, a set of rules has been adopted covering

- typographical rules on indentations, etc.
- naming conventions,
- package-interface rules,
- conventions on the error handler, the handling of dynamic structures, and the mechanisms of input/output.

These detailed rules have been collected by Rimvall (*1986*). To clarify the notations used in the following examples from IMPACT, a few general rules will be mentioned:

- Reserved Ada-words (e.g. `FOR`, `LOOP` and `END`) are to be written in upper-case characters. Variable, subprogram and parameter names are generally to be written in lower-case letters.
- Types, variables and subprograms exported from an Ada package must take the prefix `IMxy_`, where `xy` is a two lower-case letter package abbreviation.
- Variables declared within subprograms take the prefix `l_` to denote their local nature.
- Formal subprogram parameters take the prefix `p_`.

5.3 Error handling in CACSD

In interactive programs, such as the here treated CACSD packages, two basic groups of errors occur during execution of the program – **operating errors** due to errors or inconsistencies in the user input, and **programming errors** due to errors in the IMPACT program itself.

In command-driven interactive programs such as IMPACT, user-generated **operating errors** are unavoidable. These can be of a syntactical nature, e.g. when the user enters

```
WHILE i <> 5 LOPP ...
```

where `<>` has been used instead of `/=` for inequality, and `LOOP` is misspelled, or of a semantical nature as in the assignments

```
A = [1,2,3,4];
B = A(1,2)
```

where the one-dimensional array `A` is referenced with two indices. In both cases, the execution of the entered command must be

```
B = A(1,2)
        ^ **** ERROR
%I-USER-ERROR, Wrong number of indices.
%I-MESSAGE, You have specified too few or too many
%I-MESSAGE, indices during access to the variable "A".
```

Figure 5.1: Typical error message after an operating error.

halted to display a comprehensive error message of the form shown in Figure 5.1. This error message contains all pertinent components of an operating error report:

- an indication where the error occurred.
- a short, poignant error message.
- further information to resolve ambiguities or indicate hidden relations (in our example the indication that the error occurred while accessing `A`).

After an operating error has been reported to the user, the program should resume normal execution, for example by waiting for further input from the user. Under no circumstances may an interactive program "crash", as then all interactively created data would be lost. Neither may the program enter undefined states leading to an unreliable further behaviour. Instead, the recovery from all operating errors must be made to *one* predictable state, from which the user always knows how to continue to operate the program.

Although the ultimate goal of all software design is to construct error-free programs, such goals still remain utopian. It is true that modern languages such as Ada have been designed to minimize the "bug-rate" (number of errors per 1000 lines of code). The rate of both design- and coding-errors tend to decrease considerably when using modern design and programming tools (e.g. Ada instead of FORTRAN); nevertheless, precautions should be taken to minimize the effects of remaining programming errors.

```
%I-IMPACT-ERROR, An illegal attempt to deaccess a main
%I-IMPACT-ERROR, reference to a string of type
%I-IMPACT-ERROR, IMba_private_string has been made.
%I-IMPACT-ERROR, Still existing secondary references
%I-IMPACT-ERROR, blocked this deaccess.
%I-MESSAGE, The string was "SELF-GENERATED-ERROR".

%I-MESSAGE, This is a programming error in IMPACT.
%I-MESSAGE, Please submit a report to the IMPACT-manager
%I-MESSAGE, Include a diary-file of the action leading
%I-MESSAGE, to the error. Quote the exact error-name,
%I-MESSAGE, which is :BA_I_STR_ILL_DEACC_SEC_EXIST

%TRACE - IMPACT_basic               $$  IMba_deaccess
%TRACE - IMPACT_math_arithmetic     $$  Add
%TRACE - IMPACT_math_arithmetic     $$  IMma_dual_operations
%TRACE - IMPACT_kernel_execute      $$  E_dual_operation
%TRACE - IMPACT_kernel_execute      $$  E_expression
%TRACE - IMPACT_kernel_execute      $$  E_assignment
%TRACE - IMPACT_kernel_execute      $$  E_statements
%TRACE - IMPACT_kernel_execute      $$  IMke_execute
%TRACE - IMPACT_kernel_parser       $$  PA_until_error
```

Figure 5.2: Typical error message after a run-time error caused by a programming error.

As with the **operating errors**, a run time error caused by some programming error must result in a comprehensive error message. Contrary to the operating errors, these messages are normally not to be interpreted by the user (although he must be told that something went wrong), but to be passed on to the software manager of IMPACT for corrective actions. Therefore, the content of such a message centers on the type of the error, where it occurred, and under which circumstances. A typical report is shown in Figure 5.2. A traceback is included to help the programmer find the bug. These kind of error-messages tend to become rather voluminous, but they should hopefully not appear too often anyway!

The implementation of a comprehensive and consistent error-handler for both operating- and programming-errors throughout a large program is not a trivial task. In the following two sections, we will discuss the detection of and recovery from errors in general. Thereafter, we will present the error-handler of IMPACT.

5.3.1 The detection of errors

Most of the conventional programming languages, such as FORTRAN or Pascal, contain no standard language elements for the detection of run-time errors. Certain compilers/run-time-systems offer a dynamic error-handler through the use of so-called "traps", but these implementations are all system dependent. Hence, the only way to handle run-time errors in a portable manner in conventional programming languages is through *defensive programming.* This implies a cumbersome precautionary testing on error-conditions (e.g. with `IF` statements) prior to the execution of any operation which could lead to an error. For example, numerical algorithms would have to contain a test for zero denominators before each division, making the programs notably larger, definitely slower, and certainly harder to read.

To free programmers from such test-oriented defensive programming and yet let them ensure that their programs will not crash in a zero divisions, etc., many computers allow for so-called "traps". Thereby, any illegal operation is caught, and the program resumes execution in some user-defined error-handling routine. Unfortunately, such trap-oriented programming is highly system dependent, and it is not always possible to catch the errors "locally", that is, to remain in the routine where the error occurred.

In Ada, language-integrated **EXCEPTIONS** take care of the problem of detecting and, as we will see later, recovering from run-time errors. The needs for language-integrated schemes for handling errors have been discussed for some time (*Goodenough, 1975*), and exception-handlers implemented on top of existing languages havs been implemented for example for real-time applications (*Maier, 1984*). However, Ada is the first widely spread language that includes exception-handling as a standard language feature.

```
  PROCEDURE Ada_1(...) IS
  BEGIN
    --
    -- Body of the procedure
    --
  EXCEPTION
    WHEN NUMERIC_ERROR =>
      --
      -- Do something against the occurred numeric error
      --
 END Ada_1;
```

Figure 5.3: Program skeleton including exception-handler for catching all numerical run-time errors.

Ada defines six standard exceptions which will be *raised* automatically in cases of numerical under-/overflow, illegal access to non-existing dynamic structures, task errors, etc. Moreover, the user can define further exceptions of his own, and raise them when error-conditions occur. As soon as an exception has been raised, the execution of the present procedure/function is continued at the exception-section of the subprogram. If no exception section has been defined in the subprogram where the error occurs, the control is passed on to the exception section of the calling subprogram, and so forth. Hence, the procedure outline shown in Figure 5.3 will let the program recover from a numeric error within the procedure and return normally to the calling routine. If another kind of error occurs, this is propagated to the calling routine.

5.3.2 Recovery from errors

In the procedure of Figure 5.3, we assumed that the recovery from a numerical error could be made locally. In many cases, this is not possible. For example, after a division-by-zero by the execution of the user input

```
  XX = 0;
  A = [9/XX, 3*XX; 1, 3]
```

```
PROCEDURE Pascal_2(...;
                  VAR p_err : BOOLEAN);
  LABEL
    99;
  VAR
    error : BOOLEAN;
BEGIN
  p_err := FALSE;
  FOR i := 1 TO 10 DO
   BEGIN
     do_something_dangerous(...,err);
     (* this routine may return with the error
        parameter err=TRUE *)
     IF ( err ) THEN GOTO 99;
     do_something_else(...);
   END;
99:
  IF ( err) THEN
   BEGIN
     (* local error-recovery and error propagation *)
     p_err := FALSE;
   END;
END;
```

Figure 5.4: Local error recovery and error propagation programmed in Pascal.

we wish to halt not only the division, but also the creation of the matrix A. Therefore, certain errors must be propagated to or through calling routine(s) until a level has been reached where normal execution can continue (in our case the level where a new statement is processed). Moreover, in each of the routines that are recursively called during the illustrated operation, local dynamic structures may have been created/accessed. These must be deaccessed through error-recovery actions local to each routine.

In conventional languages, the local recovery from and further propagation of error conditions must be performed over error-parameters and jumps to the end of each routine. Figure 5.4 illustrates how this compels the Pascal programmer to employ "non-structured" programming elements such as GOTO's.

In languages supporting exception-handling, both the local

```
PROCEDURE Ada_2(...) IS
BEGIN
  FOR i IN 1 .. 10 LOOP
    do_something_dangerous(...);
    -- This routine may propagate the exception USER_ERROR_1
     do_something_else(...);
  END LOOP;
EXCEPTION
  WHEN USER_ERROR_1 =>
    -- Insert local recovery from error number 1 here.
    RAISE USER_ERROR_1;
END Ada_2;
```

Figure 5.5: Local error recovery and error propagation recovery programmed in Ada.

recovery and further propagation can be formulated in a structured and at the same time much more compact manner, as shown in Figure 5.5. Note the absence of error-handling in the actual body of the routine, this separation of "normal" code from error-handling code makes the program easier to construct, understand, and maintain. As we will see in the next section, it also enables a standardization of the error-handling throughout larger programs.

5.3.3 The exception-handler of IMPACT

In an interactive software package with the size and complexity of IMPACT, the inclusion of a consistent error-handler is imperative. Moreover in IMPACT, the mixture of local/global dynamic structures with secondary access (as will be described later) requires an error-handler with both local error recovery and recursive error propagation. To keep the implementational efforts of such an error-handler within limits, an approach has been taken where the exception section of every procedure/function within IMPACT is highly uniform. Also, the collection and display of error-reports is made by one central set of routines.

```
PROCEDURE Ada_3(...) IS
BEGIN
  --
  -- body with further calls to other subprograms.
  --
EXCEPTION
  WHEN IMba_exception_propagation =>
    IMba_trace_error(package_xy,"Ada_3");
    RAISE IMba_exception_propagation;
  WHEN OTHERS =>
    IMba_report_other(package_xy,"Ada_3");
    RAISE IMba_exception_propagation;
END Ada_3;
```

Figure 5.6: Minimal exception of each routine in IMPACT.

Using this streamlined approach, all subprograms of IMPACT are protected by the same error-handler with little implementational overhead. Although the exception-handler constitutes some 10-20% of the IMPACT code, it is so uniform that its implementation time can be estimated to less than 5% of the total programming effort. This error handler is activated by the first occurrence of an error; it collects traceback and auxiliary information during the entire error recovery period, and finally, it displays a comprehensive error message when a stable state has been reached. Thereafter, the error facility is deactivated until the next error occurs.

In simple routines, where no local recovery from errors is needed and where no special error-conditions need to be tested, the exception-handler displayed in Figure 5.6 suffices. The declared exception, `IMba_exception_propagation`, is propagated through each level of subprogram calls until the level is reached where normal execution can be resumed. On each level, the procedure `IMba_trace_error` is called to append the subprogram name to the trace-list. Thereafter, the exception is propagated to the next level of call. If an unexpected error should occur in the body of procedure `Ada_3` of Figure 5.6, the `WHEN OTHERS` section of the exception-handler is executed. As the procedure is supposed to be error-free, we just report that some unknown (`OTHER`) error has occurred, and propagate the exception.

If there is a foreseeable chance that, for example, a certain tasking-error occurs within a routine, an additional exception section WHEN TASKING_ERROR should be included to handle and report this error. This is one of the cases illustrated in subprogram Ada_4 of Figure 5.7. The name of the reported error, XY_U_tasking_no_process, follows a convention whereby the first two characters of the error-name correspond to a standard two-letter abbreviation of the name of the IMPACT package where the error occurred (e.g. the abbreviation KP for the package IMPACT_kernel_parser), the fourth character indicates whether this is an operating (User) or programming (Impact) error, and the remainder uniquely identifies the exact error-condition. In a central error-file, an entry with this name must also exist.

To enhance the usefulness of the error-handler to the advanced user who wishes to include his own Ada algorithms into the package, additional error-files may be added to the system. Moreover, to insure consistency between the errors declared in Ada code and the error messages given in the error-files, an algorithm which checks each error-report in the Ada source-code against all error-files has been developed.

Whenever the programmer needs a more detailed error report than one of the six predefined Ada-exceptions, or when he wishes to check whether a certain illegal state resulting in an Ada exception has occurred, he has two possibilities (Figure 5.7).

- If no special local error-recovery is necessary, the programmer would test for the error-condition and call one of the routines IMba_propagate_error. In this routine, the given error-name is stored away in the dynamic error-tree and the IMba_exception_propagation is raised to halt the execution of the body.

- If the programmer needs to include some special error-recovery action, he would declare a local exception, raise this exception, and include a corresponding exception section (WHEN exception_extra in Figure 5.7).

In both cases, specific error-messages are read from the error-file(s) for display.

```
PROCEDURE Ada_4(...) IS
  exception_extra : EXCEPTION;
BEGIN
  --
  -- body with two examples of tests for error conditions
  --
  IF ( ... ) THEN
    RAISE exception_extra;
  END IF;
  --
  IF ( ... ) THEN
    IMba_propagate_error("XY_U_nonsquare_matrix");
  END IF;
  --
EXCEPTION
  WHEN exception_extra =>
    IMba_report_error("XY_U_extra_error",
                      package_xy,"Ada_4");
    -- local error-recovery is to be inserted here.
    RAISE IMba_exception_propagation;
  WHEN TASKING_ERROR =>
    IMba_report_error("XY_U_tasking_nonexisting_process",
                      package_xy,"Ada_4");
    -- local error-recovery is to be inserted here.
    RAISE IMba_exception_propagation;
  WHEN IMba_exception_propagation =>
    IMba_trace_error(package_xy,"Ada_4");
    -- local error-recovery is to be inserted here.
    RAISE IMba_exception_propagation;
  WHEN OTHERS =>
    IMba_report_other(package_xy,"Ada_4");
    -- local error-recovery is to be inserted here.
    RAISE IMba_exception_propagation;
END Ada_4;
```

Figure 5.7: Example combining several of the previously illustrated error-handling elements into one procedure.

```
PROCEDURE IMkp_parse_interactively
          (p_session : IN OUT IMks_private_session;
           p_stream  : IN OUT IMkr_private_input_stream) IS
  l_mode          : parsing_mode_record;

  -- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  PROCEDURE PA_until_error IS
    l_statement    : IMki_statement;
  BEGIN
    IMkr_advance_symbol(p_stream,l_mode.eol_action,
                        new_stat_prompt);
    WHILE ( IMks_continue_execution(p_session) ) LOOP
      PA_global_statement(l_mode,p_session,
                          p_stream,l_statement);
      PA_stat_terminator(l_mode,p_stream,new_stat_prompt);
      IMke_execute(p_session,p_stream,l_statement);
      IMki_deaccess(l_statement);
    END LOOP;
  EXCEPTION
    WHEN IMba_exception_propagation =>
      IMba_trace_error(package_kp,"PA_until_error");
      IMki_exception_deaccess(l_statement);
      IMkr_error_in_input_stream(p_stream);
      IMba_display_error_and_negate;
    WHEN OTHERS =>
      IMba_report_other(package_kp,"PA_until_error");
      IMki_exception_deaccess(l_statement);
      IMkr_error_in_input_stream(p_stream);
      IMba_display_error_and_negate;
  END PA_until_error;

BEGIN -- IMkp_parse_interactively
  WHILE ( IMks_continue_execution(p_session) ) LOOP
    PA_until_error;
  END LOOP;
EXCEPTION
  WHEN IMba_exception_propagation =>
    IMba_trace_error(package_kp,"IMkp_parse_interactively");
    RAISE IMba_exception_propagation;
  WHEN OTHERS =>
    IMba_report_other(package_kp,"IMkp_parse_interactively");
    RAISE IMba_exception_propagation;
END IMkp_parse_interactively;
```

Figure 5.8: Main procedure of the parser implementing the error-reporting and final error-recovery.

When the program-execution has been propagated back to a level where normal execution can be resumed, which often is at the position where individual user-commands are interpreted, routines are called to display the error-message and backtraces. Thereafter, the execution is continued in a normal fashion. This is illustrated in Figure 5.8, where the `WHILE` loop of the routine `PA_until_error`, which reads and interprets user input, is executed until an error occurs. After the error-handling has been completed with a call to `IMba_display_error_and_negate`, procedure `PA_until_error` is temporarily left. However, unless an indication to halt the execution has been received, procedure `PA_until_error` is immediately called again from the outer `WHILE` loop for further parsing and interpretation of user input.

Although not illustrated here, the IMPACT error-handler also allows for the inclusion of one or more *occurrence-specific* data-fields with each error-message. This data, which must be in string form, can be dynamically added to each error-message either at the time of the original error-report, or later during the recovery period. This data is thereafter inserted in the displayed error-message according to position(s) of special characters in the predefined, generic error-message. The error-message in Figure 5.1 contained such a field, the name `A` on the last line of the message.

The IMPACT error-handler can also cope with multiple errors, for example when new errors occur during error-recovery. In this case, each error will be separately reported back to the user. This means that quite intricate structures may be created between the occurrence of a first error and the final error-report. Figure 5.9 gives a graphical view of such a dynamically created error-message tree. Note that each error-message can be accompanied by a dynamic list of trace information and another list with occurrence-specific data-fields.

To complement the error-handler, routines for the display of warnings are available. These are, for example, to be used when an error is of an intermediate nature allowing for immediate recovery, but when a message is to be passed on to the user anyway.

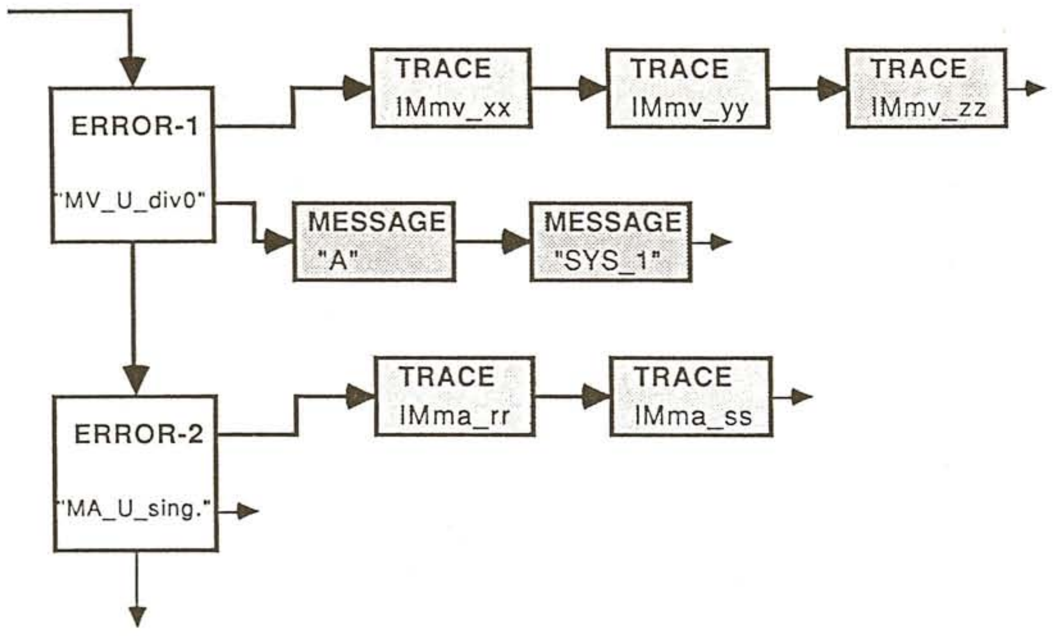


Figure 5.9: Structure of an internal error-message tree as constructed during a multiple-error recover.

5.4 Numeric data structures

As we have seen in Chapter 4, a general purpose CACSD package should support a wide range of control-related data structures. Furthermore in an interactive package, it must be possible to dynamically create new structures of these types with varying sizes as described in Chapter 3. In this section, we will show how these two requirements can be met in a very elegant manner using Ada.

One of the data structures described in Chapter 4 was the polynomial matrix with complex coefficients. This data structure can be thought of as a three-dimensional structure with the matrix dimensions in the first two and the coefficients of each polynomial in the third dimension. In a structured language like Pascal, we could define a general three-dimensional array with some maximum dimensions, and use this definition each time we needed a variable of type polynomial matrix as shown in Figure 5.10 New structures of this kind could thereafter be dynamically created using the statement

```
local_poly := NEW non_fac_poly_matrix;
```

Note that we waste memory space each time the needed matrix is smaller than the maximum allowed size.

In FORTRAN, there are no provisions for dynamically created data structures. However, due to the weak typing of FORTRAN, a common solution to these kind of problems is to map all intricate structures onto one-dimensional array(s), and construct appropriate formulas to access the different elements:

```
DIMENSION RNFPM(100000),TNFPM(100000)
```

where the real and imaginary components are stored in two different arrays. Individual coefficients of our nonfactorized polynomial matrix can be accessed via the formula

```
IPOS = IX + IY*IYDIM + (IP+1)*IPDIM*IYDIM + IOFSET
RNFPM(IPOS) = ...
```

```
TYPE
  IMba_complex = RECORD
                   re,
                   im   : REAL;
                 END RECORD;

  non_fac_poly_matrix =
      ARRAY[1..max_m_dim,1..max_m_dim,0..max_poly_dim]
            OF IMba_complex;

  nfpm_pointer = ^non_fac_poly_matrix;

VAR
  local_poly : nfpm_pointer;
```

Figure 5.10: Declarations for nonfactorized polynomial matrices in Pascal.

where IX, IY and IP are the three positions within the polynomial matrix, and IOFSET points to the beginning within the stack of the data structure in question. This technique, which is often referred to as garbage collection technique, and which corresponds to the method used in MATLAB, guarantees some dimensional flexibility (IXDIM*IYDIM*(IPDIM+1)<=1000), but a program using this scheme will be far less readable and more error prone than the aforementioned Pascal version.

In Ada, we use **discriminants** to obtain full dimensional flexibility, *and* retain readable code. Figure 5.11 shows the parts of IMPACT defining non-factorized polynomial matrices. Similar declarations are used for all numerical structures. However, certain declarations can be kept simple by re-using previously declared structures. For example, the matrix definitions are reused in linear system-descriptions.

Using the declared type IMmt_nf_poly_matrix_record (see Figure 5.11), it is possible to create polynomial matrices of *arbitrary* dimensions. However, we do not only want to define polynomial matrices statically, but we also want to dynamically create new structures of this type during the execution of the interactive program. Hence the last line in Figure 5.11 declares

```
--
-- Declarations in package IMPACT_basic
--

TYPE IMba_float     IS -- SYSTEM DEPENDENT FLOATING TYPE USED
                       -- THROUGHOUT THE IMPACT PACKAGE.

TYPE IMba_complex  IS RECORD
                           re,
                           im : IMba_float := 0.0;
                         END RECORD;

IMba_zero_complex  :  CONSTANT IMba_complex := (0.0,0.0);
IMba_unity_complex :  CONSTANT IMba_complex := (1.0,0.0);

--
-- Declarations in package IMPACT_basic
--

TYPE IMmt_gen_nf_poly_mat IS ARRAY (IMmt_positive_dim RANGE <>,
                                    IMmt_positive_dim RANGE <>,
                                    IMmt_natural_dim  RANGE <>)
                                    OF IMba_complex;

TYPE IMmt_nf_poly_matrix_record
             (x_dim,
              y_dim        : IMmt_positive_dim;
              deg          : IMmt_natural_dim) IS
            RECORD
              val          : IMmt_gen_nf_poly_mat(1 .. x_dim,
                                                  1 .. y_dim,
                                                  0 .. deg);
            END RECORD;

--  All coefficients of nonfactorized polynomials are stored
--  using indices from 0 to the order of the polynomial.
--  Spaces are reserved according to the maximum order.

TYPE IMmt_nf_poly_matrix IS ACCESS IMmt_nf_poly_matrix_record;
```

Figure 5.11: Declarations for nonfactorized polynomial matrices illustrating discriminants and dynamic structures of Ada.

```
FUNCTION New_poly(p_x,p_y : IN IMmt_positive_dim;
                  p_deg   : IN IMmt_natural_dim)
                  RETURN       IMmt_nf_poly_matrix IS
BEGIN
  RETURN NEW IMmt_nf_poly_matrix_record(p_x,p_y,p_deg);
END New_poly;
```

Figure 5.12: The creation of dynamic structures with varying sizes.

an access-type to the polynomial matrix (corresponding to a pointer in PASCAL). We *could* now, at any time, create new dynamic polynomial matrices of arbitrary sizes, as in Figure 5.12. However, due to the lack of a standardized garbage-collector in IMPACT, we will use a more general dynamic data handler as described in the next section. In any case, the advantage of working with dynamic discriminated variables is clear; at no time do we allocate more memory space than required, and there is no upper dimensional limit.

The use of structured types not only saves space, but also enhances the readability of Ada programs considerably. Another feature extensively used in IMPACT to increase program self-documentation is **subprogram overloading**. Using the above definition of the type IMba_complex, we can define functions operating on complex numbers as shown in Figure 5.13. We can now operate on variables of type IMba_complex using the normal operators as in:

```
PROCEDURE Nonsense IS
  a_c, c_1, c_2, d_2 : IMba_complex;
BEGIN
  a_c := ( c_1 + c_2 ) * d_2;
END Nonsense;
```

In this manner, we can overload the basic arithmetic operations and other IMPACT functions for all applicable numerical as well as non-numerical data structures. We thereby obtain a very simple and modular program structure, using a minimum of different function names.

```
FUNCTION "+"(p_a,
             p_b : IN IMba_complex)
             RETURN   IMba_complex IS
BEGIN
  RETURN (p_a.re+p_b.re, p_a.im+p_b.im);
END "+";

FUNCTION "-"(p_a,
             p_b : IN IMba_complex)
             RETURN   IMba_complex IS
BEGIN
  RETURN (p_a.re-p_b.re, p_a.im-p_b.im);
END "-";

FUNCTION "*"(p_a,
             p_b : IN IMba_complex)
             RETURN   IMba_complex IS
BEGIN
  RETURN (p_a.re*p_b.re - p_a.im*p_b.im,
          p_a.re*p_b.im + p_a.im*p_b.re);
END "*";

FUNCTION "/"(p_a,
             p_b : IN IMba_complex)
             RETURN   IMba_complex IS
  l_p,
  l_q:  IMba_float;
BEGIN
  IF ( ABS(p_b.im) > ABS(p_b.re) ) THEN
    l_q := p_b.re / p_b.im;
    l_p := p_b.im + p_b.re * l_q;
    RETURN (((p_a.re * l_q + p_a.im)/l_p),
            ((p_a.im * l_q - p_a.re)/l_p));
  ELSE
    l_q := p_b.im / p_b.re;
    l_p := p_b.re + p_b.im * l_q;
    RETURN (((p_a.re + p_a.im * l_q)/l_p),
            ((p_a.im - p_a.re * l_q)/l_p));
  END IF;
END "/";
```

Figure 5.13: Overloaded basic arithmetic operations on the type IMba_complex.

5.5 Maintaining dynamic structures

Typically, the designer of interactive programs has a hard time estimating the nature and size of the problems future users will try to solve. Using conventional programming languages and traditional programming approaches, the implementor would therefore arbitrarily choose some maximum limits on different entities of the program (for example maximum dimensions or maximum expression complexity) and hope that no user would ever need "more". Therefore, even programs advertised as having "no upper limits" will hit the ceiling if only the problem to be solved is big enough. It is then of little consolation to the user that, after he has lost all his interactively created data, he can enlarge the allowed size of his problem by changing some parameters/constants governing the memory-allocation and recompiling the whole program.

In IMPACT, we have taken another approach to the problem of dimensional and operational limits. IMPACT is the first CACSD package to fully adhere to the $0/1/\infty$ principle, which means that all limitations of the package are given by:

- 0 — Not allowed,
- 1 — One unit is allowed (e.g. only one keyboard may be used for the input to any one session) or
- ∞ — Any number/size is allowed (e.g. any number of characters in a name/string)

As a consequence of our decision not to limit the size and/or number of variables, sessions, and plots in IMPACT, all interactively created entities correspond to dynamically created Ada-structures with discriminated sizes. As the inclusion of a run-time "garbage-collector" for dynamically created structures is optional even in validated Ada-compilers (!), and as even programs on virtual-memory machines will eventually run out of memory if no dynamic structures are reused, IMPACT also contains a garbage-collector of its own for all reusable dynamic structures.

As IMPACT supports a large number of different dynamically created and deaccessed data structures, ranging from numerical data over text strings to symbolically stored differential equations, a standardized scheme for creating, manipulating, and recycling structures has been developed. This scheme, which will be discussed in detail in the next three sections, also utilizes the Ada-concept of information hiding for higher programming security.

5.5.1 Single-access structures

In a package of IMPACT's size, it is normally impossible for any one programmer to keep up-to-date on the details of all available data-structures and their correct employment. Therefore, any implementation where a programmer can perform manipulations on all available data structures in an indiscriminate fashion inevitably leads to error-prone code. The creators of Ada realized this and introduced the concept of "hidden" or **private types**. Variables of these types can be accessed freely only within the package (module) where they are defined. In all other parts of the program, only operations supported by exported procedures can be performed. This increases programming security without limiting generality, as the programmer can define any number of subprograms to perform all necessary operations.

There is yet another reason for limiting the visibility of *dynamic* data structures. While it is undisputed that structured programming in general leads to robuster software, we should not be led to believe that all Pascal programs are robuster than corresponding FORTRAN programs. Pascal supports dynamic structures and pointers, and although these features have become indispensable for anybody developing packages such as IMPACT, the shuffling of pointers to different dynamic entities can result in so unstructured and error-prone programs that some computer scientists advise people *not* to use pointers at all. To counter this danger, a scheme to make "pointer programming" safer will be presented in this and the next section.

In the single-access version of the IMPACT dynamic memory-management scheme described here, the dynamic structures are defined locally within a package with only a LIMITED PRIVATE access-type being exported. As limited private variables have unaccessible internal components and moreover may not be used in assignment statements, this solves both the information hiding and the pointer problem (with the drawback that each dynamic structure may be accessed by only one pointer).

This approach also places all memory-maintenance/garbage-collection routines within the body of the package in a standardized fashion. Hence, only the type declaration, three routines for creating and recycling variables, and some routines for manipulating variables are exported. To guarantee maximum garbage-collection, the recycling routines must be called for each variable as soon as it is no longer needed.

In the following case-study, we will discuss the implementation of a dynamic structure IMxy_demo. The declarations in the specification-part of the package are shown in Figure 5.14. Only one LIMITED PRIVATE type is exported, as the information in the PRIVATE part is accessible only to the compiler. Hence, the user of this package can create, deaccess, i.e. recycle over the garbage-collector, copy and manipulate dynamic structures over procedure and function calls only. In particular, implementational details of IMxy_demo are not exported. This gives the programmer of the package the freedom to change internal implementational details at wish as long as he does not change the visible behaviour of the package.

In the body of the package, the internal structure is declared as shown in Figure 5.15. We here assume that the declared structure contains a static element number as well as a possible access to further dynamic entities some_access. The element next_demo is needed by the garbage-collector. The access element unused_demos_start is used by the garbage-collector to store recycled and presently unused variables.

Figure 5.16 shows a garbage collector for the IMxy_demo structure. The code shown here is somewhat simplified. In reality, these procedures also contain some statements for collecting statistics. Sometimes, instantiations of generic templates are used. In New_demo, we test whether an appropriate structure can be found in unused_demo_start which is then extracted; if not, a new dynamic structure is created. In Recycle_demo, we

```
  --
  -- Exported part of the package specification.
  --
  TYPE IMxy_demo IS LIMITED PRIVATE;


  PROCEDURE IMxy_create_demo       (p_demo : IN OUT IMxy_demo;
                                    ...);
  --
  PROCEDURE IMxy_deaccess          (p_demo : IN OUT IMxy_demo);
  PROCEDURE IMxy_exception_deaccess(p_demo : IN OUT IMxy_demo);
  --
  PROCEDURE IMxy_copy_second_and_deaccess
                                   (p_new,
                                    p_old  : IN OUT IMxy_demo);
  PROCEDURE IMxy_copy_second       (p_new  : IN OUT IMxy_demo;
                                    p_old  : IN     IMxy_demo);
  --
  PROCEDURE IMxy_manipulating_demo (p_demo : IN OUT IMxy_demo;
                                    ...);
  --
  -- Private part of the package specification.
  --
PRIVATE
  TYPE demo_record;
  TYPE IMxy_demo IS ACCESS demo_record;
```

Figure 5.14: The exported specification part for managing a single-access dynamic structure.

return an existing dynamic structure to the garbage collector. If further dynamic structures are accessed by the returned entity, we recycle these, too.

Figure 5.17 shows templates of three of the standard exported procedures for creating and recycling the dynamic data structures. IMxy_exception_deaccess is to be called from every exception-handler where local variables or parameters of type IMxy_demo have been declared. These structures are then deaccessed if they exist, but no error is raised if they do not exist (which is the case in IMxy_deaccess). The procedures performing the copying have not been shown in detail. Thereby, the

```
--
-- Declarations in the package body.
--
TYPE demo_record IS RECORD
                      number       : predefined_scalar_type :=1;
                      some_access  : predefined_access_type;
                      next_demo    : IMxy_demo;
                    END RECORD;

unused_demos_start : IMxy_demo;
```

Figure 5.15: Elaborations of the limited-private type defined in the previous figure. This declaration is placed in the package body.

procedure IMxy_copy_second_and_deaccess copies pointer values using the active statements

```
p_new := p_old;
p_old := NULL;
```

whereas IMxy_copy_second makes a physical copy of the whole structure, and therefore does not change p_old. Both procedures contain tests to ensure that p_old exists and p_new does not exist at the time of the call.

5.5.2 Multiple access structures

A major advantage can be gained through liberal use of dynamic structures with frequent copying of pointer-values, namely that the number of copies of intricate (dynamic) structures in different physical storage-locations can be kept to a minimum, saving both CPU-time and memory. However, such copying of pointer-values is always dangerous as one dynamic structure (one physical storage location) then may be referenced several times over different pointers, giving the programmer ample opportunity to produce "pointer-spaghetti". Moreover, any garbage-collector can easily be fooled by recycling elements having concurrent

```
-------------------------------------------------------
FUNCTION New_demo RETURN IMxy_demo IS
  l_demo : IMxy_demo;
BEGIN
  IF ( unused_demos_start = NULL ) THEN
    l_demo              := NEW demo_record;
  ELSE
    l_demo              := unused_demos_start;
    unused_demos_start  := l_demo.next_demo;
    l_demo.next_demo    := NULL;
  END IF;
  RETURN l_demo;
EXCEPTION
  WHEN STORAGE_error =>
    IMba_report_error("IM_L_no_dynamic_space",
                      package_xy,"New_demo");
    RAISE IMba_exception_propagation;
  WHEN IMba_exception_propagation =>
    -- Normal exception-handler
END New_demo;


-------------------------------------------------------
PROCEDURE Recycle_demo(p_demo : IN OUT IMxy_demo) IS
BEGIN
  IF ( p_demo.some_access /= NULL ) THEN        -- recycle
   Recycle_something(p_demo.some_access);       -- dynamic
  END IF;                                       -- elements
  l_demo.number      := 1; -- reinitialize static elements
  p_demo.next_demo   := unused_demos_start;
  unused_demos_start := p_demo;
  p_demo             := NULL;
EXCEPTION
  -- Normal exception-handler
END Recycle_demo;
```

Figure 5.16: Garbage collector for a single-access dynamic structure.

```
-------------------------------------------------------
PROCEDURE IMxy_create_demo  (p_demo : IN OUT IMxy_demo;
                             ...) IS
BEGIN
  IF ( p_demo /= NULL ) THEN
    IMba_propagate_error("XY_I_demo_exist");
  END IF;
  p_demo := New_demo;
  -- transport additional input parameter
  -- to the new structure.
EXCEPTION
    -- Normal exception-handler
END IMxy_create_demo;


-------------------------------------------------------
PROCEDURE IMxy_deaccess(p_demo : IN OUT IMxy_demo) IS
BEGIN
  IF ( p_demo = NULL ) THEN
    IMba_propagate_error("XY_I_demo_nonexist");
  END IF;
  Recycle_demo(p_demo);
EXCEPTION
-- Normal exception-handler
END IMxy_deaccess;


-------------------------------------------------------
PROCEDURE IMxy_exception_deaccess
               (p_demo : IN OUT IMxy_demo) IS
BEGIN
  IF ( p_demo /= NULL ) THEN
  Recycle_demo(p_demo);
  END IF;
EXCEPTION
-- Normal exception-handler
END IMxy_exception_deaccess;
```

Figure 5.17: Body of the main routines for handling single access dynamic structures.

pointers (causing logically different structures to occupy the same physical memory location). This section will describe a scheme which retains the flexibility of "pointer programming", and yet guarantees a proper handling in terms of an early detection of most errors involving pointers.

As it is not possible to "both keep and eat the candy", no software scheme can be totally error-preventive, AND allow the programmer full freedom to manipulate every pointer. The scheme presented here leaves the programmer fairly much freedom, for example by allowing multiple access to dynamic variables, at the cost of a somewhat delayed error detection. As any inconsistencies are detected and reported by the garbage collector, the scheme is ideal for interactive programs where dynamic data structures have generally a short "half-time". In such programs, the garbage-collector will detect any logical/programming error early enough, causing an error-message to be passed on together with a trace (see Figure 5.2).

As in the single-access case, the IMPACT implementation of this scheme uses the `LIMITED PRIVATE` type-concept of Ada. Thereby, the exported interface as shown in Figure 5.18 is identical to the single-access case with only one additional subprogram, `IMxy_access_second`. The `PRIVATE` part reveals that each `IMxy_demo` is not implemented as an access variable any more, but as a record (this of course is of no consequence for the programmer using the package). In this record, `ref_kind` classifies the access by taking the value `nonexisting_structure`, `main_reference` or `secondary_reference`. The element `ref` corresponds to `IMxy_demo` in the single-access case.

The main idea behind this multiple-access dynamic memory-manager is as follows: each dynamic structure has a **main** reference (main access pointer). As in the single-access case, this main reference may only be copied from one variable to another using the procedure `IMxy_copy_second_and_deaccess`, thus allowing for only one main reference per physical storage location. In addition to the main reference, several so-called **secondary** references to the same structure may be created through calls to `IMxy_access_second`. These structures must also be deaccessed through calls to `IMxy_deaccess`.

To ensure a consistent pointer-handler, the exported subpro-

```
  --
  -- Exported part of the package specification.
  --
  TYPE IMxy_demo IS LIMITED PRIVATE;


  PROCEDURE IMxy_create_demo        (p_demo : IN OUT IMxy_demo;
                                     ...);
  --
  PROCEDURE IMxy_deaccess           (p_demo : IN OUT IMxy_demo);
  PROCEDURE IMxy_exception_deaccess(p_demo : IN OUT IMxy_demo);
  --
  PROCEDURE IMxy_access_second      (p_new,
                                     p_old  : IN OUT IMxy_demo);
  PROCEDURE IMxy_copy_second_and_deaccess
                                    (p_new,
                                     p_old  : IN OUT IMxy_demo);
  PROCEDURE IMxy_copy_second        (p_new  : IN OUT IMxy_demo;
                                     p_old  : IN     IMxy_demo);
  --
  PROCEDURE IMxy_manipulating_demo (p_demo : IN OUT IMxy_demo;
                                     ...);
  --
  -- Private part of the package specification.
  --
PRIVATE
  TYPE demo_record;
  TYPE demo_access IS ACCESS demo_record;
  TYPE IMxy_demo   IS
            RECORD
              ref_kind  : IMba_reference_enum
                                := nonexisting_structure;
              ref       : demo_access;
            END RECORD;
```

Figure 5.18: Specification part of a multiple-access dynamic structure.

```
--
-- Declarations in the package body.
--
TYPE demo_record IS
        RECORD
         number       : predefined_scalar_type := 1;
         some_access  : predefined_access_type;
         nbr_of_refs  : NATURAL := 0; -- internal "bean-
                                      -- counter" of
                                      -- secondary references.
         next_demo    : demo_access;
          END RECORD;

unused_demos_start : demo_access;
```

Figure 5.19: Elaborations of the limited-private type defined in the previous figure. This declaration is placed in the package body.

grams shown in Figure 5.18 must be used according to the following rules:

- the main reference MAY NOT be deaccessed as long as any further (secondary) references to the variable exists. To ensure such a consistency, hidden "bean-counters" are employed. These bean-counters are implemented as integer elements of the main dynamic structure, as illustrated in Figure 5.19. The bean-counters are incremented in `IMxy_access_second`, and either decremented or tested upon in `IMxy_deaccess`, as shown in Figure 5.20.

- as in the single-access case, it is not permitted to directly replace an already existing reference with another reference. Therefore, all routines setting pointer values, such as the routine `IMxy_access_second` in Figure 5.20, contain a corresponding test.

- to avoid data-consistency problems, it is recommended that only a limited number of actions are made available for secondary referenced structures, for example by

```
PROCEDURE IMxy_access_second(p_demo : IN OUT IMxy_demo;
                             p_old  : IN     IMxy_demo) IS
BEGIN
  IF ( p_demo.ref_kind /= nonexisting_structure ) THEN
    IMba_propagate_error("XY_I_demo_exist");
  END IF;
  IF ( p_old.ref_kind = nonexisting_structure ) THEN
    IMba_propagate_error("XY_I_demo_nonexist");
  END IF;
  p_demo.ref_kind := secondary_reference;
  p_demo.ref      := p_old.ref;
  Increment(p_old.ref.nbr_of_refs); -- bean-counter increment
EXCEPTION
  -- Normal exception-handler
END IMxy_access_second;

-------------------------------------------------------
PROCEDURE IMxy_deaccess(p_demo : IN OUT IMxy_demo) IS
BEGIN
  CASE p_ref.ref_kind IS
    WHEN nonexisting_reference =>
      IMba_propagate_error("XY_I_demo_nonexist");
    WHEN main_reference =>
      IF ( p_demo.ref.nbr_of_refs > 0 ) THEN
        IMba_propagate_error("XY_I_demo_secondary_exist");
      END IF;
      p_demo.ref_kind := nonexisting_structure;
      Recycle_demo(p_demo.ref);
    WHEN secondary_reference =>
      Decrement(p_demo.ref.nbr_of_refs); -- bean-counter
      p_demo.ref_kind := nonexisting_structure;
      p_demo.ref      := NULL;
  END CASE;
EXCEPTION
  WHEN IMba_exception_propagation =>
    IMba_trace_error(package_xy,"IMxy_deaccess");
    p_demo.ref_kind := nonexisting_structure;
    p_demo.ref      := NULL;
    RAISE IMba_exception_propagation;
  WHEN OTHERS =>
  -- Corresponding local error-recovery
END IMxy_deaccess;
```

Figure 5.20: Body of some exported routines for handling multiple-access dynamic structures.

```
IF ( p_demo.ref_kind = secondary_reference )  THEN
  IMba_propagate_error("XY_I_demo_ill_oper_on_sec");
END IF;
IF ( p_demo.ref_kind = main_reference ) AND THEN
   ( p_demo.ref.nbr_of_refs > 0 ) THEN
  IMba_propagate_error("XY_I_demo_ill_oper_sec_exist");
END IF;
```

Figure 5.21: Tests to be included in all exported manipulation-subprograms to implement a "read-only" rule.

> treating these as "read-only" entities. Depending on the nature of the structure, it might even be advisable to apply the same limitation on main references with further secondary references. The implementation of these limitations is easily performed by including the tests shown in Figure 5.21 in all applicable manipulation routines of the package where the private type is defined.

If the main program includes a shut-down procedure deaccessing ALL dynamic structures (in IMPACT, this is made over a handful of recursive deaccess-calls), all consistency errors in the handling of pointers will be detected not later than when the package is exited.

The here described scheme to increase program security and robustness through the use of private types can be made mandatory in all packages of a project but one — the very package where the private type is declared. Within this package, which also is where the garbage-collector resides, all "normal" pointer operations are allowed — and for some basic manuipulation even necessary. It is, however, recommended that the exported, robust routines are used also within this package wherever possible. Anyhow, since most logical errors are created when programmers are unfamiliar with some data- or program-structures, the error-rate should be lower inside this package.

Apart from unique manipulation routines, the following standard routines are normally also supplied for all dynamic structures:

- `IMxy_structure_exists` to indicate whether a (private) dynamic structure exists or not.

- `IMxy_get_main_reference`. If a given reference is secondary, this procedure call will copy the whole structure (using `IMxy_copy_second`), and return the new main reference (with the secondary reference deaccessed).

5.6 Ada as a command-language interpreter

The flexible command-language and data-structures described in Chapters 3 and 4 constitute a powerful user interface to the IMPACT package. In the IMPACT project, these specifications were the result of a first feasibility study made well before a single line of Ada-code existed. However, as with all interactive packages, such an initial *definition* must also be followed by a tedious *implementation* of a parser actually interpreting and executing the user-input. As suggested by Wirth (*1977*) and others (*Aho et al., 1986*), a simple but yet general implementation includes a scanner to decode the input into basic terminal symbols, and a parser to interpret these symbols. Apart from the pure language-theoretical parsing-problem, the actual implementation of an immediate and yet efficient parser/interpreter deserves additional attention. Thereby, several approaches are possible:

- The conceptually simplest, but executably slowest, approach is *direct interpretation* which can also be denoted as pure *data-driven* programming. Input strings are thereby traversed and interpreted. The scanner groups individual characters into primitive terminal symbols (tokens) such as `FOR`, `1.3` and `=` and passes these on to the parser. The parser then interprets the input, and directly calls the routines executing the entered command. This method is acceptable for command-languages where an entered line is interpreted one time only (as e.g. most operating systems commands). Whenever an entered command, or a section thereof, is executed in a repetitive fashion, direct interpretation becomes very slow as sections of the input strings have to be scanned/interpreted over and over again.

- On the other extreme, a purely *code-driven* implementation will scan and parse the entered input as described above. However, rather than directly interpreting the commands, a compiled version of the commands is created, and this compiled code (machine code) is then brought to execute. Sometimes this transformation to compiled code is made over several steps; the model description of many simulation languages, for example, are translated into another higher-level language (mostly FORTRAN) before they are further compiled into assembly code. Although this initial translation requires more time than the direct interpretation, the execution of the code once compiled will be faster by several orders of magnitude. This approach therefore pays off when structures need to be executed repetitively. However, this approach is either not interactive, or the interactive program has to invoke compilers/linkers *and* incorporate the linked program into itself to enable interactive execution with consistent data (introducing extremely system-dependent self-modifying code as in *Essebo, 1981*).

- Between these two extremes, any number of combinations exist. One example of such a compromise is the *"threaded-code"* (*Loeliger, 1981; Korn, 1982*) approach. Thereby, the input is scanned into terminal symbols by the scanner, and the parser translates these symbols into internal structures (for example dynamic tree-structures) corresponding to the logical structure of the input commands. This approach enables the parser to take care of the syntactical and of most of the semantical analysis of the input. The internal structures are then brought to execution by a special interpreter which will attain a resonable speed as only part of the semantical analysis is left to be performed at run time, i.e. "in the loop".

In IMPACT, the threaded code approach has been chosen as a reasonable compromise between implementational complexity, execution speed, and portability. Although the code-generation approach is the fastest, providing at least a factor of ten in speed-up as compared to the threaded-code approach, that approach was considered too system-dependent to warrant further investigation. However, an off-line automatic compiling facility is planned for IMPACT which will accept soft-coded subprograms, and preprocess them into hard-coded Ada-programs. These Ada programs adhere to a format consistent with the hard-coded section of IMPACT, and can therefore be compiled and linked together with the IMPACT package. Thereby, the desired speed-up in terms of execution time will be gained at the cost of a more time-consuming compilation-link cycle. Hence, this method is be primarily used to transform tested and stable subprograms from their soft-coded to a hard-coded form.

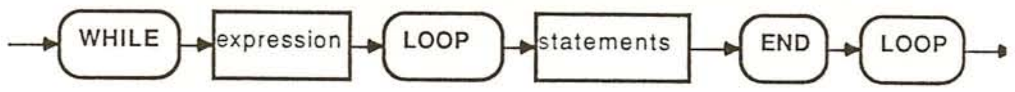
As already mentioned in Section 3.6, the syntax of IMPACT belongs to the LL(1)-class of grammars (*Aho et al., 1986*). The main reason for such a grammar was the availability of an LL(1)-oriented general-purpose parser (*Bongulielmi et al., 1984*) with which we could define the grammar of the IMPACT command language (see Appendix 1) and check its consistency and completeness before we started to implement the parser of IMPACT. Moreover, this general purpose parser allowed us to test whether example programs adhered to the just defined grammar, and thus gave us a tool to check early on the expression power of the command language, and the suitability of the language for describing control-related structures and algorithms.

As a consequence of choosing an LL(1) grammar, we could construct a recursive descent parser without lookahead facility. Such a recursive descent parser is extremely simple, and fairly efficient (as long as we do not use a computer where subprogram calls are unproportionally time-consuming, in which case the high level of recursion takes its toll).

Together with the similarity between the IMPACT command-language and Ada, the choice of a LL1(1) syntax and a recursive descent parser made the parser-structure extremely simple. Let us study the Ada-code needed to parse and interpret the `WHILE` loop of the input sequence in Figure 5.22, which adheres to the LL(1)-syntax in Figure 5.23. The task of the parser is to interpret the `WHILE` structure correctly, and create the internal

```
i := 0;
WHILE ( i > 9 ) LOOP
  i := i + 1;
END LOOP;
```

Figure 5.22: Example of IMPACT input to be treated.



```
while_loop = 'WHILE' expression 'LOOP'
                statements
             'END' 'LOOP' .
```

Figure 5.23: Syntax of an IMPACT `WHILE` loop.

IMPACT-code (ICODE) shown in Figure 5.24.

This ICODE is made up of dynamic statement and expression tree elements connected in a fashion corresponding to the hierarchical structures of the entered commands. In Figure 5.25, we see that the parser-section has a straight forward structure with recursive calls to other routines for all non-terminal symbols supplemented with external checks of all terminal symbols. The thus created ICODE is passed on for execution as soon as a complete statement has been parsed. Figure 5.26 shows the `WHILE` section of the executional part of IMPACT. We note the extreme simplicity obtained by mapping the `WHILE` loop of the user-input onto an Ada `WHILE` loop with recursive calls for the *condition* evaluation and *statements* execution.

The definition of the IMPACT command-language consists of over 40 production rules (see Appendix 1). For a syntax of such a relatively large size, the parser and executing part of IMPACT are quite compact. The parser consists of approximately 3000 lines of code, and the executing part has less than 2000 lines, in-

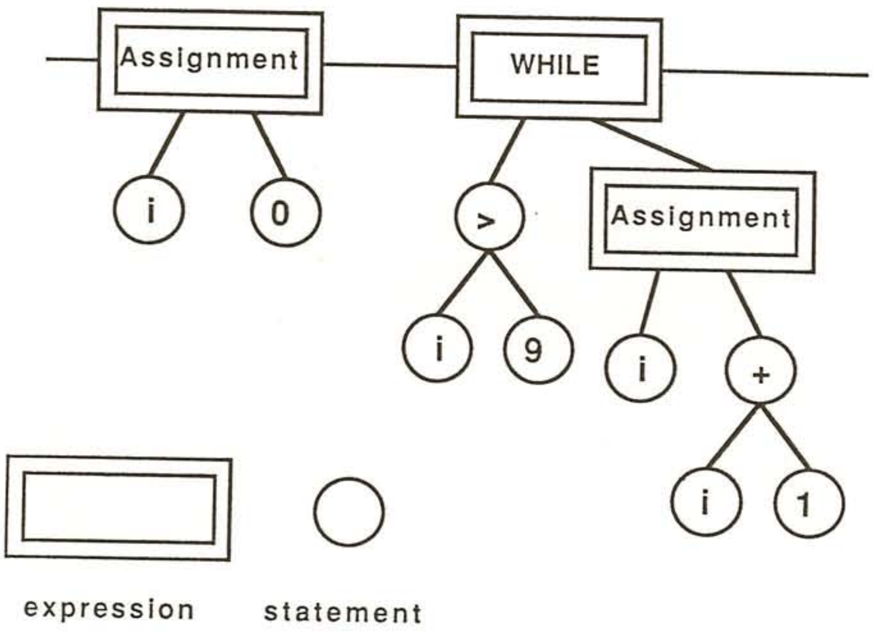


Figure 5.24: ICODE to be constructed form the previous sample input.

cluding a large percentage of standardized error-handling code. Together, they make up less than 10% of the IMPACT kernel-system. This can be attributed to the simplicity of an LL(1) syntax, and the power of a recursive descent parser with all current parsing-information "built into the recursion".

A drawback of the current approach compared with a solution using e.g. an LR(1) syntax with a table-driven parser is the relative difficulty of including good error-handler and, in particular, error-recovery mechanisms in the parser (*Amman, 1978; Poplawski, 1978*).

The simplest of approaches to error handling is taken in the present version IMPACT – the parser will detect and report only the first error occurring in the user input, without attempting to parse any further symbols. However, as the input to the parser

```
PROCEDURE PA_while_loop
             (p_mode   : IN     parsing_mode_record;
              p_stream : IN OUT IMkr_private_input_stream;
              p_stat   : IN OUT IMki_statement) IS
--
--  This routine implements the production **while-loop**
--
BEGIN
  IMki_create_statement(p_stat,p_stream,while_statement);
  IMkr_advance_symbol(p_stream,p_mode.eol_action);
  PA_expression(p_mode,p_stream,p_stat.first_expression);
  IF ( NOT IMkr_test_reserved_word(p_stream,word_loop) ) THEN
     IMba_propagate_error("KP_U_while_loop_expct");
  END IF;
  IMkr_advance_symbol(p_stream,p_mode.eol_action);
  PA_statements(p_mode,p_stream,p_stat.first_substatement);
  IF ( NOT IMkr_test_reserved_word(p_stream,word_end) ) THEN
    IMba_propagate_error("KP_U_while_end_loop_expct");
  END IF;
  IMkr_advance_symbol(p_stream,p_mode.eol_action);
  IF ( NOT IMkr_test_reserved_word(p_stream,word_loop) ) THEN
    IMba_propagate_error("KP_U_while_end_loop_expct");
  END IF;
  IMkr_advance_symbol(p_stream,return_eol_flags);
EXCEPTION
  WHEN IMba_exception_propagation =>
    IMba_trace_error(package_kp,"PA_while_statement");
    RAISE IMba_exception_propagation;
  WHEN OTHERS =>
    IMba_report_other(package_kp,"PA_while_statement");
    RAISE IMba_exception_propagation;
END PA_while_loop;
```

Figure 5.25: Part of the parser decoding a while-loop.

```
PROCEDURE E_while_statement
             (p_info       : IN     executing_info_record;
              p_session    : IN OUT IMks_private_session;
              p_statement  : IN     IMki_statement) IS
 l_while_value : BOOLEAN;
BEGIN
  E_boolean_expr(p_info,p_session,
            p_statement.first_expression,l_while_value);
  WHILE ( l_while_value ) LOOP
    E_statements(p_info,p_session,
                 p_statement.first_substatement);
    E_boolean_expr(p_info,p_session,
              p_statement.first_expression,l_while_value);
  END LOOP;
EXCEPTION
  WHEN IMba_exception_propagation =>
    IMba_trace_error(package_ke,"E_while_statement");
    RAISE IMba_exception_propagation;
  WHEN OTHERS =>
    IMba_report_other(package_ke,"E_while_statement");
    RAISE IMba_exception_propagation;
END E_while_statement;
```

Figure 5.26: The part of the interpreter executing a WHILE loop.

is mainly interactive with a normal unit to be parsed having a size of only one or at the most a few lines, the drawback of this error-recovery mechanism was not considered to be all too serious. Moreover, the implementation of a more comprehensive error handler by manually stacking away backtracking information (as in the case of a table driven parser) would be possible. Thereby, for normal tree parsing, the recursion mechanism would still be used, while for error recovery an alternative table-driven scheme implemented within the exception-handling sections of IMPACT would attempt to recover from the error. This, however, would create a certain implementational as well as executional overhead. Alternatively, ongoing research by other groups might soon result in a mechanism for automatically describing and implementing error-recovery mechanisms for LL(1) languages (e.g. *Lewi et al., 1976, 1983*).

5.7 The modularity of IMPACT

In large software projects, the coordination of different subprojects has proven to be an extremely difficult task (*Sommerville, 1985*). More than one large program has been completely rewritten because nobody knew his way around the interwoven program structures and data-flows any more. Attacking this problem, Ada, more than any other programming languages, actively supports a modular overall design of large programs. The resulting modularity of the code will be of greatest advantage not only during the programming phase, but also during the testing and maintenance phases. For general purpose CACSD packages such as IMPACT, this means for example that each group of algorithms can be developed independently, and with a minimum of interconnections to nonalgorithmic parts.

Presently, the IMPACT kernel-system contains some 35 packages divided into the 8 main groups as depicted in Figure 5.27. These groups also have internal hierarchies of their own so that only one or two of the individual packages of each group ever need to be accessed from the outside.

This overall modularization has proven very advantageous for the IMPACT development group which has included a large number of persons, each working on a particular part of the package for a shorter period of time (typically students working during one semester of 14-16 weeks). In such an environment, each new co-worker must be able to work effectively within a week or so, a goal which was possible to attain with the present modular construction of IMPACT. The ideal result of e.g. one semester project has been a finished package to be used as a tool during subsequent projects.

As stated previously, the advantages of modularization do not limit themselves to the initial development phases. As IMPACT is intended to be an open kernel-system, it will also be possible for the general user to add new algorithms to it, algorithms that he needs in his own application. With the functional parti-

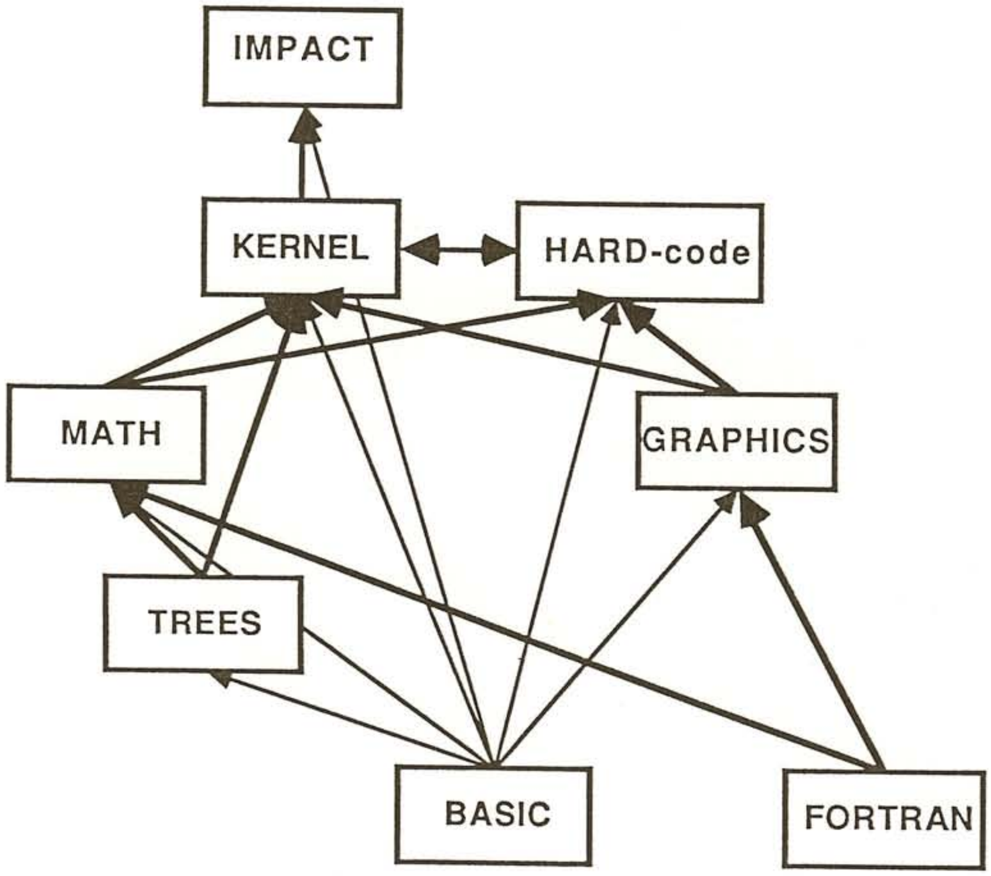


Figure 5.27: A global view of the packages constituting the IMPACT project.

tioning of the IMPACT packages and package groups, the user incorporating a new Ada (or even FORTRAN) hard-coded algorithm typically needs to access subprograms in only half a dozen different packages, or less than 20% of all IMPACT packages.

As a case study, let us consider the incorporation of a new hard-coded subprogram in IMPACT on the systems level (any incorporation on the user-level is made in exactly the same manner, but in package `IMPACT_hard_user`). Normally, the following steps will be taken:

- To describe the user-interface of the new subprogram, a declaration following the same syntax as for soft-coded subprograms is entered (see Figure 5.28). The analogy be-

```
FUNCTION LINCONT(P_A,
                 P_B,
                 P_C : IN MATRIX;
                 P_D : IN MATRIX = 0) RETURN STATE_SPACE
?BEGIN?
?GI? Creates a continuous linear state-space description
     out of four correctly dimensioned matrices according
     to the formulae
?NA?   .
       x = A*x + B*u
       y = C*x + D*u

?AJ? where x denote the state vector, u the input signal
     and y the output signal.
     Example of a legal call:
?NA?
       SYS1 = LINCONT(mya, myb, myc)

?PN? P_A
?PS? A-matrix or system matrix (no default)
?PI? System matrix of the linear continuous system
     description, which is the matrix A in the formulae
?NA?   .
       x = A*x + B*u
       y = C*x + D*u

?AJ? The dimension of this matrix must be (N,N) where
     N is the number of states of the system (equal to
     the dimension of the x-vector).
?PN? P_B
?PS? B-matrix or input matrix (no default)
?PN? P_C
?PS? C-matrix or output matrix (no default)
?PN? P_D
?PS? D-matrix or direct matrix (DEF = Zero matrix)
```

Figure 5.28: Definition of the hard-coded subprogram LINCONT. The help information for the parameters P_B, P_C and P_D has been shortened.

```
%I-USER-ERROR, Some standard hard-coded IMPACT-subprograms no
%I-USER-ERROR, have no corresponding Ada-code in package
%I-USER-ERROR, IMPACT_hard_user.
%I-MESSAGE, The names of these subprograms are:

LINCONT
```

Figure 5.29: Error message displayed when the body of a hard-coded subprogram has not been defined.

tween declaring soft- and hard-coded subprograms makes this step trivial for anybody familiar with the interactive interface of IMPACT. The thus entered declaration, which includes parameter specifications and help-/query-information, may be checked by the IMPACT parser for correctness and consistency before it is incorporated into the library of "system subprogram headers".

- The inclusion of the header can be made without any Ada compilations or link-sequences. However, as no Ada-coded body has been incorporated, IMPACT would complain during the next startup with the message shown in Figure 5.29. Consequently, we have to add the body of our new subprogram to package `IMPACT_hard_system`. First, we complement the main enumerated type and the main case statement of the package as illustrated in Figure 5.30. Thereafter, we add the procedure `E_LINCONT` describing the action to be taken during the execution of the subprogram. Note that until now absolutely NO knowledge of other parts of IMPACT was needed, hence the work to incorporate a stand-alone hard-coded subprogram without parameters would end here.

- As normal hard-coded subprograms reference other parts of IMPACT, for example the variable-handler and numerical algorithms, the user must familiarize himself with a few package specifications. In particular, a strong familiarity with `IMPACT_basic` is indispensable, as this package contains not only the error-/exception-handler but also a

```
TYPE subprogram_enum IS (check_error,
                         ..
                         ..
                         length,
                         lincont, -- *** NEW ENTRY **
                         lindiscr,
                         ..
                         ..);


PROCEDURE IMhs_execute_subprogram
           (p_session     : IN OUT IMks_private_session;
            p_environment : IN OUT IMks_private_environment;
            p_subprogram  : IN     NATURAL) IS
  l_subprogram : subprogram_enum;
BEGIN
 l_subprogram := subprogram_enum'VAL(p_subprogram);
 CASE l_subprogram IS
  WHEN check_error =>E_check_error(p_session,p_environment);
  ..
  ..
  WHEN length   =>   E_length(p_session,p_environment);
  --
  -- *** NEW ENTRY ***
  WHEN lincont  =>   E_lincont(p_session,p_environment);
  -- *** NEW ENTRY ***
  --
  WHEN lindiscr =>   E_lindiscr(p_session,p_environment);
  ..
  ..
 END CASE;
EXCEPTION
  -- Regular exception-handler.
END IMhs_execute_subprogram;
```

Figure 5.30: Compulsory changes to the body of the procedure IMPACT_hard_system when a new subprogram (LINCONT) is defined.

general name-/string-handler and several output-facilities.

- For each extraction or insertion of a parameter-value, a subprogram of `IMPACT_kernel_session` is called as in Figure 5.31. Also for the handling of numerical values, some types found in `IMPACT_math_types` must be used together with the dynamic data-handler of `IMPACT_math_variable`. Most likely, the developed subprogram uses some numerical algorithm as primitives in which case one or several procedures from other mathematical packages must be used.

- The thus modified package `IMPACT_hard_system` can now be compiled and linked to the IMPACT package. In the cases where the actions can be described in the interactive IMPACT command-language, the planned compilation unit described in Section 5.6 will perform all of the here described actions automatically, including the automatic change to the hard-coded environment routines shown in Figure 5.30..

In this case-study, we have seen that a user can add new hard-coded algorithms having detailed knowledge of only 5–6 of the 35–40 hitherto implemented packages of IMPACT. However, during normal execution of thus created subprogram, typically 50% or more of all IMPACT packages are involved, including the full parser/executor, display and plot routines, different garbage-collectors and the (unavoidable) error-handler routines. Our experience shows that, although the same kind of simple/adaptive interfaces to hard-coded subprograms *could* have been implemented in any other computer language, the modularization of Ada gave us this simple interface more or less "for free".

5.8 The robustness of Ada-code

In discussions on the advantages and disadvantages of Ada, one aspect is mostly forgotten, namely the sturdy syntax of the language which is far robuster than the syntax of any other language known to the author (including German!). This is best

```
PROCEDURE E_LINCONT
        (p_session     : IN     IMks_private_session;
         p_environment : IN OUT IMks_private_environment) IS
  l_a,
  l_b,
  l_c,
  l_d,
  l_result         : IMmv_private_math_variable;
BEGIN
  IMks_access_variable(p_environment,"P_A",l_a);
  IMks_access_variable(p_environment,"P_A",l_b);
  IMks_access_variable(p_environment,"P_A",l_c);
  IMks_access_variable(p_environment,"P_A",l_d);
  --
  IMmb_build_cont_system(l_result,l_a,l_b,l_c,l_d);
  --
  IMmv_deaccess(l_a);
  IMmv_deaccess(l_b);
  IMmv_deaccess(l_c);
  IMmv_deaccess(l_d);
  IMks_save_return_chain(p_environment,l_result);
  --
EXCEPTION
  -- Regular exception-handler with "exception-deaccess"
  -- of the local dynamic structures l_a .. l_result.
END E_LINCONT;
```

Figure 5.31: Example of a body implementing a hard-coded subprogram. The actual numerical algorithm is to be found in subprogram IMmb_build_cont_system, which is exported from package IMPACT_math_build.

illustrated by an example (*Bucher, 1984*). A well known FORTRAN bug involved the omission of a comma in a FORTRAN DO loop (an error involving only one single character). This simple error transforms the DO loop into an assignment statement without having the compiler produce even a warning message:

```
C*****INCORRECT FORTRAN    (*Equivalent Pascal*)
      SUBROUTINE CACSD     Procedure CACSD;
      DO 999 I=1 100         Var DO999I : Integer;
      CALL CONTR           Begin
  999 CONTINUE               DO999I := 1100;
      RETURN                 CONTR;
      END                   End;
```

In *this* case, Pascal seems to have a robuster syntax than FORTRAN, something which advocates of Pascal like to think is generally true. However also in Pascal, small lexical errors can change the meaning of a program entirely:

```
(*Correct Pascal*)           (*Incorrect Pascal*)
   Procedure CACSD;             Procedure CACSD;
     Var i : Integer;             Var i : Integer;
   Begin                        Begin
     For i := 1 To 100            For i := 1 To 100
      DO Contr;                    DO; Contr;
   End;                         End;
```

To the left, we see a correct Pascal version of the original FORTRAN program. To the right, we have deliberately produced another "one-character error", and voilà, we obtain *exactly* the same incorrect program we had in FORTRAN. On the other hand, Ada has a syntax which is much more robust:

```
-- Equivalent Ada               -- Correct Ada
PROCEDURE CACSD;                PROCEDURE CACSD;
BEGIN                           BEGIN
  FOR i IN 1 .. 100 LOOP          FOR i IN 1 .. 100
    NULL; END LOOP; Contr;          LOOP Contr; END LOOP;
END CACSD;                      END CACSD;
```

In Ada, there is no risk that a similar error will occur as the differences between the two versions are much larger (an "11 character error" must be made). This is true not only for the above example; in general, the code of Ada is much more robust than that of FORTRAN or Pascal. For example, no undeclared variables or implicit statements may exist (except for `FOR LOOP` counters which always are implicitly declared and thereby only locally available), and each structural `END`ing specifies what it ends (`END LOOP`, `END Chapter`).

5.9 Conclusions

In this chapter, we have studied different implementational aspects of CACSD software and, in particular, the suitability of the Ada language for interactive CACSD packages. As the results in this chapter have not been obtained in some feasibility study of limited scope, but during the development of an actual CACSD-kernel of some 60 000 lines of code with a coding effort exceeding 4 man-years, our general assessment is that it can be quite conclusively said that the general suitability of Ada for large CACSD projects is *excellent*. For a complete evaluation, however, the negative sides of Ada should also be mentioned together with suggestions on how the more serious consequences of these drawbacks can be avoided:

- The use of Ada in a larger software project is certainly a rupture with the old tradition of implementing every scientific program in FORTRAN, in particular, since practically all existing CACSD or general numeric algorithms are FORTRAN coded. However, ADA-libraries of algorithms are expected to emerge on the market in the near future. Therefore, IMPACT contains a well-defined interface for later incorporation of new algorithms.

- As FORTRAN will remain the main implementation language for control algorithms for some time, IMPACT must be able to access these algorithms as well. Fortunately, the developers of Ada have realized that this would be a problem for all initial Ada projects, and thus have included the `PRAGMA` concept by which it is possible to call subprograms written in other languages in a well-defined manner. In IMPACT, these pragmas have been concentrated into three small packages (one each for system calls, FORTRAN-coded algorithms and the graphical package).

- During the next couple of years, only a few Ada compilers of yet unknown quality will exist. Therefore, Ada users risk serving as guinea pigs for the compiler constructor. This is certainly a valid objection against Ada, and therefore a trade-off has to be made between estimated loss of time and money during the first implementation, and the gain in the long run (in particular in the maintenance and update stages). However, for exactly this reason, it is probably correct that, in an industrial rather than academic environment, we would not have been able to convince our management to embark on such a venture. Also, despite the expected future long-term maintenance record of programs implemented in Ada, companies presently could have problems maintaining their Ada-products (!). Currently, there are primarily FORTRAN programmers on the market to be hired. Thus, any program coded in another language faces the company with a serious training problem in a world where the average "company life span" of a software engineer may be below two years (at least in the USA).

- More than in any other structured language, the code of Ada-programs gets voluminous due to declarations, exception handlers and type conversions. In IMPACT, this is countered by the above described standardization of the error-handler and dynamic memory-managers which shorten the implementation time and help decreasing the error-rate.

These critical aspects of Ada are, however, outweighed by a multitude of indisputable advantages. Although some of these features partially exist in other languages as well, Ada is the only language bringing them all under the same roof:

- Ada allows arbitrary types of data-structures to be directly defined, avoiding the hazzle of redefining all structures into arrays (the only structure available in FORTRAN). Furthermore, through the use of discriminants, Ada allows for the dynamic sizing of arrays which means that no unnecessary space has to be reserved, as would be the case in a language like PASCAL.

- Ada, due to recursiveness, allows for a much more elegant coding of the IMPACT expression parser than FORTRAN would do.
- Ada is highly structured, making modular programming possible, resulting in reliable and easily maintainable code. Furthermore through the use of visibility rules, all implementational details and system-dependencies can be hidden from the user as well as from most of the people involved in the development of IMPACT.
- Ada is per definition portable, there may not exist sub- and/or super-sets of Ada with that name. Therefore, IMPACT shall be better maintainable and easier to port to new computers than most other CACSD-packages.
- Ada provides for a unique means of exception handling. The main difference between the Ada exception handler and most conventional (user defined) error handlers is that Ada can handle user-errors (e.g. erroneous interactive input sequences) as well as system-/programming-errors (e.g. division by zero or array-index out of range) in the same portable manner.
- Together with the mentioned Ada features, the robust syntax of Ada will ensure less error prone programs than what could have been obtained using FORTRAN or even Pascal.

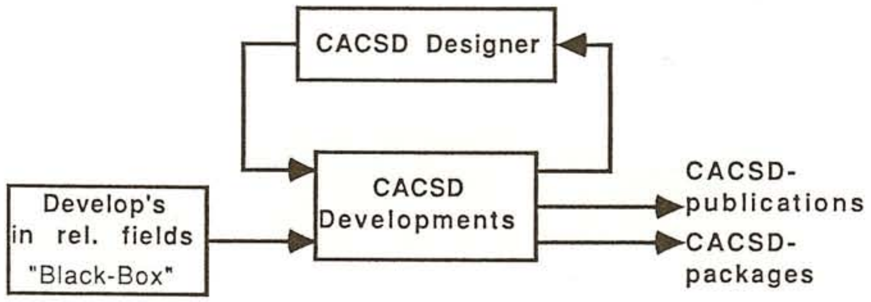
Chapter 6

FUTURE DIRECTIONS IN CACSD

As noted already in Chapter 2.1, the evolution of present-day CACSD tools has been highly dependent upon the technological advances in other, related fields. Of the concepts presented in this thesis, many would not have been implementable five years ago. Moreover, most of these concepts have originated from cross-fertilization between requirements from the control-side and catalytic ideas from other application fields. Several important conclusions can be drawn from this:

- Control engineering is a small speciality compared to the comprehensive fields of electrical engineering, computer science and computer engineering. Thus, the mentioned cross-fertilization takes place on a one-way street, with the CACSD designer sitting on the end of the road selecting bits-and-pieces from the enormous flow of new knowledge and products. Speaking in control terms, this makes the future of CACSD uncontrollable, as one of the inputs to the CACSD-process stems from an autonomous black-box (Figure 6.1). Many issues mentioned in this chapter will therefore remain pure wishes until the big black-box delivers the right signals.

- It is for us impossible to quantify the speed of development in the fields surrounding CACSD. However, we do notice that the *rate of impact* of these general developments has increased during recent years as control-programs have



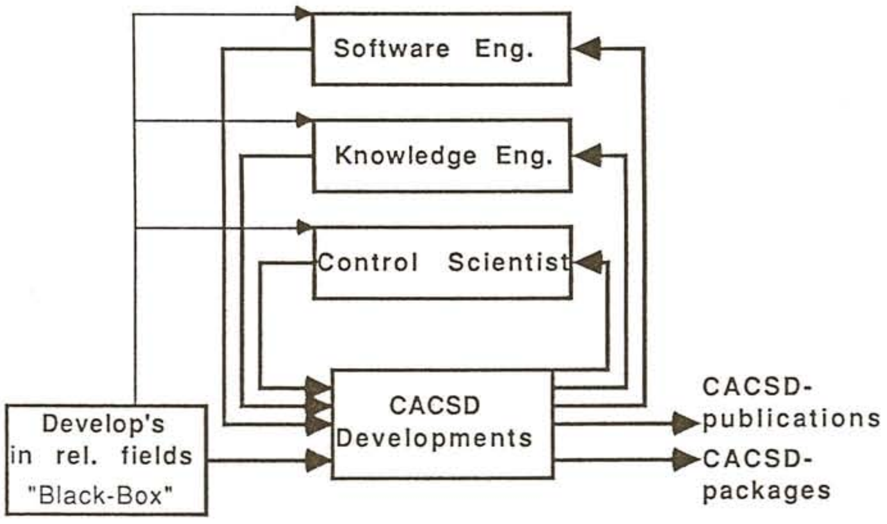


Figure 6.1: The control problem equivalence of making predictions on future trends in CACSD.

seized to be "pure special-purpose programs". The designers of modern interactive control environments are more susceptible to technological changes in the computer field than a reluctant FORTRAN-programmer was only a few years ago. Returning to our control model, we have an optimal controller (the CACSD designer) trying to perform his best with an increasing amplitude of the signal from the black-box.

- While early control packages were developed by control engineers with some programming experience, the development of modern control-tools requires much more distributed expertise. Control theory knowledge must be combined with experience in numerical software, formal language design/compiler construction (for the command language), graphical software, data-base theory (to handle voluminous and inhomogeneous data), and modern programming languages (for an adequate implementation). Due to the sheer size of a modern CACSD environment, the overall design will only be successful if modern software engineering principles are used. Needless to say, few individuals can handle all this simultaneously. Hence, our illustrative control system now becomes distributed with the software-engineer acting as coordinator.

This chapter was given the innocent title "Future Directions of CACSD". It should now be clear that the content of such a chapter is an *estimate* of the output from an *uncontrollable* MIMO-system with a partial, *distributed* controller. Only with the help of the futuristic IMPACT-package can we even attempt such an estimation ...

6.1 Graphics in CACSD

The use of graphical output (step responses, Bode plots, and so on) has become an indispensable part of all modern CACSD packages. In a student environment, the frequent use of plots to verify results and give the students an intuitive feel for systems behaviour is imperative, and in the basic courses, where everything still can be "calculated by hand", it is often the only reason for using a computer at all. Unfortunately, most packages only support special graphs adapted to the basic algorithms of the package, giving the user little freedom in constructing his own problem-adapted plots. This hampers the design sequence where critical information often can be obtained by comparing different curves or mathematically manipulated plot data (plot state `X1` versus state `X2`, plot the difference of output signals `Y1` and `Y2` versus the time, plot an error signal using a logarithmic scaling). Here, the packages CTRL-C and IMPACT are especially flexible, using their inherent data structures for the description and manipulation of graphs. Moreover, CTRL-C offers the user an interactive, "GKS-like" (*ANSI, 1985*) plot interface for a very general plot facility.

While graphical *output* is employed in almost all CACSD packages, the use of graphical *input* is still in its infancy. As we have seen in Section 3.3.1, only few packages allow for a graphical input. Nevertheless, most of these few already available packages offer the user particularly natural-to-use and easy-to-learn user interfaces. Thus, the user with little experience in CAD and scarce knowledge of control theory is greatly assisted by the use of self-explanatory graphical symbols rather than intricate descriptions using formulae or program structures. When larger systems can be entered using a hierarchical approach (with zooming), the graphical input becomes particularly interesting.

On modern engineering workstations supporting windowing techniques, a very nice combination of a graphical input facility, an alphanumerical algorithmic (command-driven) environment, and an efficient form driven and/or "pull-down menu" driven operating interface rendering a highly flexible, integrated CACSD package can be obtained. Concurrent sessions and multiple windowing can be used to separate these modes which are incompatible on most other terminals. However, the widespread use of graphical input is hampered by the hardware presently available:

- The connection of graphical input devices such as joysticks and mice is seldom foreseen by standard graphical terminals. No standard for these devices exists.
- Graphical terminals with high resolution and a fast refresh rates are needed. Ideally, bit-mapped displays with intelligent graphical processors as found on graphics workstations such as the Apollo Domain, Sun or MicroVAX-GPS should be used. On these displays, the refresh rate is fast enough for zooming and deletion/insertion of intricate, hierarchical structures.

The required graphical facilities, together with the needed computational power, dictate the hardware requirements for CACSD applications. Modern workstations provide excellent graphics and at the same time put enough computing power (in the magnitude of a standard VAX-11/780 unit) on the user's desk - an ideal environment for the dedicated control engineer. However, the price per computer-connection is rather high for the workstations and comparable to that of a multi-user minicomputer environment (another very common configuration for CACSD applications). Engineering workstations cost today between \$20.000 and \$40.000 depending on the fanciness of the graphics required, but already the next generation of workstations should be tagged at below \$10.000. New developments in workstation clustering allow for resource sharing that brings the prices further down in a multi-workstation environment. In the educational process, where literally hundreds of students have to be given access to computers for their (control-) exercises, more inexpensive solutions have to be sought. Economical alternatives are the modern personal computers (e.g. IBM-PC or MacIntosh). These provide the user with decent graphics and enough computing power for introductory to intermediate control exercises. Moreover, packages usable in control exercises have already emerged for this type of equipment (e.g. the new PC version of MATLAB, *Moler et al., 1985*).

6.2 Standards in CACSD

With the advent of interactive CACSD-packages, data is entered and algorithms are called through interactive conversation with the control package, and new algorithms can be formed (customized) as user-written macros. Moreover, a wide-spread use of packages using an algorithmic, structured and thereby extendable, command-driven interface (MATLAB, PC-MATLAB, IMPACT, CTRL-C and MATRIX_X) can be seen as a de facto "conceptual" standard of the user interface. However, a more formal standardization is needed for several reasons:

- No control package can support *all* control structures and control algorithms. Therefore, users knowing one package should be able to switch to another package immediately or after only a short familiarization period.

- In today's control packages, a large percentage of the control algorithms are not "hard-coded" (in e.g. FORTRAN, Pascal or Ada), but implemented as "soft-coded" macros in the command-language of the package. Also, all researchers/engineers will, after a few months of use of a package, have extensive macro-libraries of "customized" algorithms of their own. To save re-implementation time, macros defined in one package should therefore run in other control packages as well (as long as the required algorithms are supported by both packages).

- As previously stated, no control package will ever be able to support *all* algorithms. Consequently, a user may be forced to utilize several different control packages in solving one particular task (e.g. using different packages for the modeling, identification, analysis, synthesis and implementation) Therefore, a user should be able to exchange data from one control package to another without having to write conversion programs.

- To encourage the exchange of "hard-coded" algorithms, the transportation of algorithmic subprograms from one package to another should be facilitated.

From these points, we deduce that a standardization is required on several levels:

- A standard, interactive, command-language is needed to enable the exchange of macros, and to unify the user interface of different packages. Such a standard can, and should, be described consistently using a set of "BNF"-productions for the syntax with textual additions for semantic details (*Bongulielmi et al., 1984*). Any standard on this level will be hard to enforce on already existing packages, but is likely to be followed by new packages (compare the development of simulation languages after the publication of the CSSL'67 language standard, *Augustin et al., 1967*).
- A standard data format (supporting all common control structures) for the exchange of data between different control packages over external files. Such a standard is easy to implement even in existing packages: as each program can have several input/output modes, one such mode can support the standard while the other modes are kept as they are.
- A standard interface to the numerical algorithms to facilitate the exchange of algorithms. This is the level where hitherto most standardization attempts have been made; recent standardization proposals for subroutine libraries have been published by the Benelux Working Group on (Control) Syftware (*WGS, 1983, 1985*). However, we are no longer only dealing with stand-alone algorithms (subroutines). In addition to the "old" parameter standardizations, we now have to specify how the algorithm is to be accessed interactively. This couples the subprogram standardization with the interactive command-language standardization, and is not totally free from implementational restrictions (e.g. the supported data structures of the implementation language and target machine limitations).

As already discussed in Section 3.12, a common external data interface for the exchange of data between different packages is imperative. However, as the set of data-structures representable internally by the program vary from one CACSD-package to another, any least common denominator of all packages is bound to be much too limited (most likely only allowing for the exchange of real matrices). A possible solution to this problem has been suggested within the IFAC Working Group on Standard in CACSD Software (*IFAC, 1986*), namely that the standard is formulated to encompass a superset of all data-structures available in all packages. These structures may be stored on external, sequential ASCII-files using predefined formats with specific markings denoting the exact structure type. Each CACSD-package can then produce files with all data structures the program supports. When reading an external file, only the data-structures supported by the package can be loaded, but as the entities of the external file are specifically typed, any external data of "unknown type" can be skipped (with a possible warning to the user that certain data was ignored).

A further important reason for obtaining standards should be mentioned: the unique position of FORTRAN in the implementation of scientific software is already threatened by other languages (Ada and Modula for algorithmic implementations, C for system programming, and Prolog and Lisp for expert system connections). Thus, several standards on the program-interface to the numerical algorithms are needed (one for each implementation language). However, we notice that the first two levels of standardization (command-language and external data exchange) are independent of the underlying implementation language. The construction of and adherence to such implementation-language independent standards may thus well counteract the increase in "entropy" of CACSD software caused by any upcoming language confusion.

6.3 Expert systems in CACSD

Studying the proceedings from recent and not-so-recent conferences on CACSD (*Cuenod, 1979; IEEE, 1983, 1985, 1986*) the "hot" topics in CACSD have changed in a quite rapid pace. In the late 70's, most contributions dealt with *methods and algorithms* as well as special purpose packages. During the first half of the 80's, the emphasis switched to the design of *user-friendly interfaces*, and other communication issues. In the last year or so, *expert systems* and control has become an increasingly popular topic at CACSD conferences, leading to an approximate parity in the number of papers about algorithms, user-interfaces and expert systems. So far, most contributions on expert systems in control have been tentative in nature with elaborations on possible impact of and feasible approach to expert system employment, but also some progress reports on already implemented intelligent packages have emerged (*Taylor and Frederick, 1984; James et al., 1985; Larsson and Persson, 1986*). Thereby, three classes of expert-system employment can be distinguished:

- The implementation of new algorithms and methods (or re-implementation of old ones) through the employment of expert systems. Clearly, these methods will not be as strongly numerically oriented as the previously used methods, but be based rather on qualitative evaluations and cognitive- and/or symbolic-reasoning. This approach looks particularly promising in areas where hitherto available packages have given no or only little assistance, such as modeling or model validation (*de Swaan Arons, 1983; Zeigler, 1985*).

- The use of expert systems to guide the user in the utilization of existing algorithms. This includes guidance on the selection of proper methods, the specification of suitable constraints, starting points, numeric parameters, etc. Such expert systems would be valuable in all phases where nontrivial or nonstandard operations must be performed, for example during the identification (*Larsson and Persson, 1986*) or design phase.

- The inclusion of expert systems within the user communication part of a package to assist the user during the operation of the program itself. This could include front-ends accepting natural-language input or a more intelligent query-feature than the one presented in this thesis. One such approach is the "command-spy" concept developed at in Lund (*Larsson and Persson, 1986*), where the "expert" will only passively scan the user input until an erroneous or inconsistent command is entered. At such a point, the expert system will start to guide the user. This approach lets the user work fast and efficiently over a normal command-langauge interface *until he needs or wants help*.

Each of these groups of expert system employment certainly opens up new dimensions to the field of CACSD, and the consequences for the users of such packages is bound to be profound. However, the changes in the implementational environment of these packages will be equally profound. While current CACSD packages have been implemented in procedural languages (FORTRAN, C, Ada, etc.), expert system components may possibly be implemented in Lisp (*Winston and Horn, 1981*) or Prolog (*Clocksin and Mellish, 1984*) (or, rather, preimplemented expert system shells, which gives the implementor a higher level tool than a direct use of Lisp or Prolog, and thus saves on implementation time). These languages require a totally different programming environment, and are, because of their totally different approach to data representations and memory management schemes, not easily connectable to classical procedural languages. However, as the vast amount of software available in procedural languages cannot comfortably be translated into either PROLOG or LISP, future CACSD programs may well be implemented in a combination of different languages. One solution to this problem is to work with two separate programs communicating over mailboxes (*Trankle, 1986*). This approach, however, is very slow. Another solution is to utilize a "common language environment" as found for example on the VAX under VMS, and on different UNIX machines where the interface between subprograms is defined at the level of the operating system rather than at the level of the individual languages. This solution, however, is extremely non-portable. The last solution, which is rather heretic among expert-system "buffs", is to base the *"control expert"* not on LISP or PROLOG, but on a normal procedural language. As many presently available expert system shells are developed in "conventional" computer languages, e.g. C, this last solution is not as unwieldy as it sounds. It would also solve the connection and communication problems mentioned previously.

Despite these implementational problems, the general interest in the control community for using expert systems (in one way or another) in CACSD is strong and widespread. We can therefore expect further interesting results in this direction in the near future, possibly leading to a completely new working environment for the CACSD package implementor as well as the CACSD package user.

6.4 Conclusions

In Figure 2.1 we illustrated how the developments in CACSD have been (at least partially) dependent upon advances in other, related fields. Such cross-fertilizations will remain in the future as well, and must be seen as vital injections to our fields. Therefore, if we would draw the same picture in 5–6 years from now, we most likely have to draw one more row for the expert-system influence, we would put some markings in the graphics row and, hopefully, we could write some entries of standardization results in the CACSD row.

Chapter 7

CONCLUSIONS

A CACSD package should ideally provide **all control engineers** with **all the computational, managerial as well as documentational tools** needed during their **routine as well as irregular work** with a **minimum of overhead**. Unfortunately, this is by most presently available CACSD software packages not the case:

- Not many packages will provide **all control engineers** with an adequate tool. The main reason for this is an inadequate flexibility of the user interface, which either assumes the user to be an expert (and thereby gives the true expert a fast and efficient tool, but makes the initial hurdle to be taken by all non-experts too high), or assumes that the user is almost an idiot (by asking him too many questions, which is great for the beginner or irregular user who likes to be lead by the hand for a while, but turns off the expert after a very short time). Moreover, neither of these extremes are adequate for the average user. To counter this problem, we have in Chapter 3 presented different schemes which, combined into one package, will cover the needs of *all* users. Our solution consists of a multi-mode conversation with a fast and flexible command language combined with an informative query-feature, forms-driven input, etc.
- Most package will not supply the user with **all the tools** he needs, but only the tools of one or two of all the steps in a complete controller design cycle, or only tools for a

certain class of operations (which for example may limit the use of a package to time-domain operations only). The main reason for this is a lack of adequate data structures, preventing the implementation of all necessary algorithms. In Chapter 4 we therefore presented a complete set of data structures as required in any all-encompassing CACSD package.

- When working with conventional CACSD packages, much time is lost **managing** data. The reasons for this is manifold: the mentioned lack of the *proper* data structures forces the user to store away data in inappropriate structures (and thus confuses the user as to the content of the structures), a lack of structurability of the user interface forces the user to save all data "in one big bucket" (and thus gives him problems of retaining the overview of his actions), and, finally, a lack of a facility for storaging away large segments of data outside the package in a structured fashion. Hence, in Chapter 3 we presented the session concept which will help the user to manage his data and thereby to increase his problem solving capability. Moreover, in Chapter 4 we presented concepts supporting dynamic, user-extendable data structures. In that chapter we also discussed different requirements of an adequate data-base facility for a CACSD package.

- In "real life" much of an engineers time is spent **documenting** and presenting results. Here a flexible graphical capability of the package will do wonders, especially if this facility allow for the presentation of control system structures as well as more conventional plots.

- Looking at the advertisements of many commercial CACSD products one may often be amazed by how easy it is to solve **routine** problems. However, in buying a package for long-term use, it is just as important to find out what **irregular** problems **can not** be solved by the package. Here the extendability of the package is of utmost importance. As most control algorithms are of numerical nature, an algorithmically extendable user interface (for example in form of the interactive, structured, command language interface presented in Chapter 3) will allow the user to quickly incorporate new algorithms (or to make slight changes to old ones) whenever needed.

- Even with the fanciest of command-language interfaces, it is always possible to find one algorithm which is not implementable on the level of an interactive command-language, as one or more algorithmic primitives are missing from the package. In this case, the user must be able to add his own, "hard-coded" algorithm to the package **with a minimum of overhead**. In Chapter 5, we presented one such "plug-in" program interface to a CACSD package, and thereby also illustrated the advantages to the package-creator as well as to the package user of using a highly modular programming language such as Ada when implementing large CACSD packages.

Looking at todays CACSD packages, even the best of them suffer from one or several of the just mentioned problems. It is, however, the sincere belief of the author that the basic concepts of the new CACSD package **IMPACT**, the framework of which has been implemented during this research projects, will provide the answer to these fundamental problems hampering the use of present-day CACSD package.

Appendix A

THE SYNTAX OF IMPACT

A.1 The syntax in EBNF

The EBNF-notation of Bongulielmi and Cellier (*1984*) is used.

```
(********************************************************************
 *                                                                  *
 *  SYNTAX DEFINITION OF IMPACT:                                    *
 *                                                                  *
 *   INTERACTIVE MATHEMATICAL PROGRAM FOR AUTOMATIC CONTROL THEORY  *
 *                                                                  *
 *  THIS LANGUAGE WAS DESIGNED BY                                   *
 *                                                                  *
 *                  MAGNUS RIMVALL                                  *
 *                  INSTITUTE FOR AUTOMATIC CONTROL                 *
 *                  SWISS FEDERAL INSTITUTE OF TECHNOLOGY (ETH)     *
 *                  CH-8092 ZUERICH                                 *
 *                  SWITZERLAND                                     *
 *                                                                  *
 *                  DEFINED APRIL     1983                          *
 *                  REVISED JANUARY   1984                          *
 *                          OCTOBER   1985                          *
 *                          SEPTEMBER 1986                          *
 *                                                                  *
 ********************************************************************)
```

```
(**********************************)
(***                            ***)
(*** Top-level productions.     ***)
(***                            ***)
(**********************************)

IMPACT                  =   { global_statement statement_terminator }
                            'EXIT' .


statements              =   { single_statement statement_terminator } .


statement_terminator    =   ( ( ';' { $ '<CR>' } )
                            | ( ',' { $ '<CR>' } )
                            | (      { '<CR>' }    ) ) .


global_statement        =   ( single_statement
                            | subprogram_definition
                            | system_definition
                            | help_statement
                            | edit_statement ) .


single_statement        =   ( assignment_proc_call
                            | multiple_assignment
                            | compound_statement
                            | for_loop
                            | if_statement
                            | null_statement
                            | return_statement
                            | while_loop        ) .

(**********************************)
(***                            ***)
(*** Statement productions.     ***)
(***                            ***)
(**********************************)

assignment_proc_call    =   ( ( IMPACT_name [ parameterlist ] '=' )
                            | '<ANSWER=>'
                            | '<PROCALL>' ) simple_expression .


compound_statement      =   'BEGIN'
                               statements
                            'END' .
```

```
edit_statement        =   'EDIT' [ '(' ( 'IDENT' | STRING ) ')' ] .


for_loop              =   'FOR' ( ( 'IDENT' 'IN' [ 'REVERSE' ]
                                                 range_or_domain )
                                | ( 'INDEX' 'IN' [ 'REVERSE' ]
                                                 index_indication ) )
                            'LOOP'
                               statements
                            'END' 'LOOP' .


range_or_domain       =  simple_expression [ '..' simple_expression ] .


index_indication      =   IMPACT_name '('
                                  { ( ':'
                                    | ( 'IDENT' [ 'REVERSE' ] ) ) $ ',' }
                                        ')' .

help_statement        =   'HELP' | '?' .


if_statement          =   'IF' boolean_expression
                              'THEN' statements
                          { $ 'ELSIF' boolean_expression
                               'THEN' statements }
                             [ 'ELSE' statements ]
                          'END' 'IF' .


multiple_assignment   =   '<' { IMPACT_name $ ',' } '>'
                                '=' IMPACT_name [ parameterlist ] .


null_statement        =   'NULL' .


return_statement      =   'RETURN' ( ( expression ) |
                                     ( '<' { expression $ ',' } '>' ) ) .


while_loop            =   'WHILE' boolean_expression
                           'LOOP'
                              statements
                           'END' 'LOOP' .
```

```
(**********************************)
(***                            ***)
(*** Expression productions.    ***)
(***                            ***)
(**********************************)

boolean_expression    =   expression .


expression            =   relation [ ( ( 'AND' [ 'THEN' ] )
                                     | ( 'OR'  [ 'ELSE' ] ) ) relation ] .


relation              =   simple_expression
                            [ ( '=' | '/=' | '<' | '<=' | '>' | '>=' )
                                simple_expression ] .


simple_expression     =   [ '+' | '-' ]
                                  { term $ ( '+' | '-' | '&' ) } .


term                  =   { factor $
                           ( '*' | '.*' | '.*.' |
                             '/' | './' | './.' |
                             '\' | '.\' | '.\.' ) } .


factor                =   [ 'NOT' ]
                          ( ( IMPACT_name [ parameterlist ] )
                          | ( '[' matrix_expression ']'  )
                          | ( '(' expression ')' )
                          | STRING
                          | real )
                            [ ''' | { '`' } ] [ '**' factor ] .


IMPACT_name           =  'IDENT' [ level_indication ]
                            [ ':' 'IDENT' [ level_indication ] ]
                            { $ '.' 'IDENT' [ level_indication ] } .


level_indication      =  '[' { simple_expression $ ',' } ']' .
```

```
parameterlist          =  '(' { ( ( ':' )
                                | ( 'HELP' | '?' )
                                | ( expression
                                  [ ( '=>' expression )
                                  | ( '..' simple_expression ) ] ) )
                              $ ',' } ')' .

(* LL1 for ( ':' ) |                              --legal in indices
           ( 'HELP' | '?' ) |                     --legal in calls
           ( expression ) |                       --legal
           ( simple_expression
                [ '..' simple_expression ] ) |    --legal in indices
           ( 'IDENT' '=>' expression )            --legal in calls      *)


matrix_expression      = { {  polynomial_expression $ ',' }
                           $ ( ';' | '<CR>' ) } .


polynomial_expression  =   ( ( simple_expression
                                 [ ( { { '^' } simple_expression  }
                                   | { '|' simple_expression  } ) ] )
                           | ( { { '^' } simple_expression }         )
                           | ( { '|' simple_expression }             ) ) .


STRING                 =   '"' '"' .


real                   =   ( 'UINTEGER' [ '.' [ 'UINTEGER' ] ]
                                        [ exponent ] ) |
                           ( '.' 'UINTEGER' [ exponent ] ) .


exponent               =   'E' [ '+' | '-' ] 'UINTEGER' .

(**********************************)
(***                            ***)
(*** Subprogram productions.    ***)
(***                            ***)
(**********************************)

subprogram_definition  =   ( function_def
                           | procedure_def ) .
```

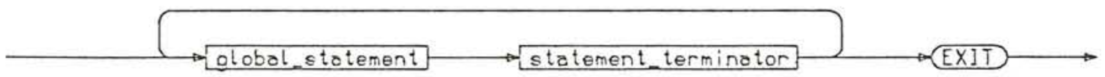
```
function_def           =  'FUNCTION'  subprogram_name
                               [ func_param_declaration ] 'IS'
                               [ help_declaration ]
                               [ variable_declaration ]
                          'BEGIN'
                               statements
                          'END' subprogram_name .


procedure_def          =  'PROCEDURE' subprogram_name
                               [ proc_param_declaration ] 'IS'
                               [ help_declaration ]
                               [ variable_declaration ]
                          'BEGIN'
                               statements
                          'END' subprogram_name .


subprogram_name        =  'IDENT' |
                          ( '<' { 'IDENT' $ ',' } '>' ) .


func_param_declaration = '(' {  { 'IDENT' $ ',' }
                                [ ':' [ 'IN' ] variable_type ]
                                [ '=' expression ]
                                $ ';' } ')' .


proc_param_declaration = '(' {  { 'IDENT' $ ',' }
                                [ ':' [ in_out_declaration ]
                                      variable_type ]
                                [ '=' expression ]
                                $ ';' } ')' .


in_out_declaration     =  ( ( 'IN' [ 'OUT' ]  )
                          | ( 'OUT' ) ) .


variable_declaration   =  { { 'IDENT' $ ',' }
                            ':' variable_type ';' } .


variable_type          =  'IDENT' .


help_declaration       =  '?BEGIN?' { anything } '?END?' .


anything               =  ( any_char | any_word ) .
```

```
any_char              =   '"' | '&' | '''   | '*' | '**' |
                          '+' | ',' | '-'   | '.' | '.*' | '.*.'|
                          '..'| './'| './.'| '.\'| '.\.'| '/'  |
                          '/='| ':' | ';'   | '<' | '<=' | '?'  |
                          '=' | '=>'| '>'   | '>='| '('  | '`'  |
                          '[' | '\' | ']'   | '^' | '|'  | ')'  .


any_word              =   'ACROSS'   | 'AND'       | 'BEGIN' |
                          'CONNECT'  | 'CONSTANT'  | 'CUT'   |
                          'EDIT'     | 'ELSE'      | 'ELSIF' |
                          'END'      | 'EXIT'      | 'FOR'   | 'HELP' |
                          'IF'       | 'IN'        | 'INDEX' | 'IS'   |
                          'LOOP'     | 'NONE'      | 'NOT'   | 'NULL' |
                          'OR'       | 'OUT'       |
                          'RETURN'   | 'REVERSE'   | 'STATE' |
                          'THEN'     | 'THROUGH'   | 'WHILE' | 'WITH' |
                          'FUNCTION' | 'PROCEDURE'| 'SYSTEM'|
                          'IDENT'    | 'UINTEGER' .


(**********************************)
(***                            ***)
(*** System productions.        ***)
(***                            ***)
(**********************************)

system_definition     =   'SYSTEM'  subprogram_name
                               system_param_decl 'IS'
                               [ help_declaration ]
                               { $ ( ( system_definition
                                       statement_terminator )
                                   | locals_declaration ) }
                               { $ cut_declaration }
                               { $ connect_declaration }
                          'BEGIN'
                               ( equations
                               | null_statement statement_terminator )
                          'END' subprogram_name .


system_param_decl     =   [ '(' { { 'IDENT' $ ',' }
                               ':' [ 'IN' ] variable_type
                               [ '=' expression ] $ ';' } ')' ]
                          [ 'IN' signal_declaration ]
                          [ 'RETURN' signal_declaration ] .


signal_declaration    =   { ( 'IDENT' |
                              ( '<' { 'IDENT' $ ',' } '>' ) )
                            ':' variable_type [ '=' expression ] ';' } .
```

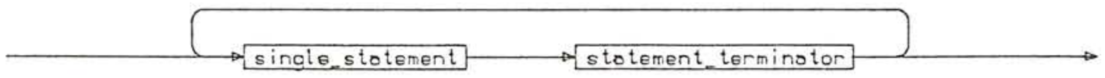
```
locals_declaration    =   { { 'IDENT' $ ',' }
                            ':' [ 'CONSTANT' | 'STATE' ]
                            variable_type [ '=' expression ] ';' } .


cut_declaration       =   [ 'NODE' ] 'CUT'
                          { cut_name '(' { { 'IDENT' $ ',' }
                                  ':' [ 'IN' | 'OUT' ]
                                  ( 'ACROSS' | 'THROUGH' | 'CUT' )
                                  $ ';' } ')' ';' } .


cut_name              =   'IDENT' .


connect_declaration   =   'CONNECT' '(' { connect_variable
                           { 'WITH' connect_variable } $ ';' } ')' ';' .


connect_variable      =   ( { 'IDENT' $ '.' }       |
                            '<' { { 'IDENT' $ '.' } $ ',' } '>' ) .


equations             =   { simple_expression '='
                            simple_expression ';' } .
```

A.2 The syntax diagrams

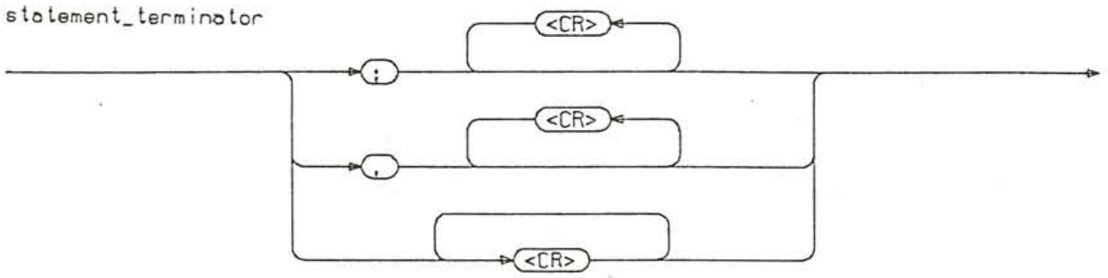
IMPACT



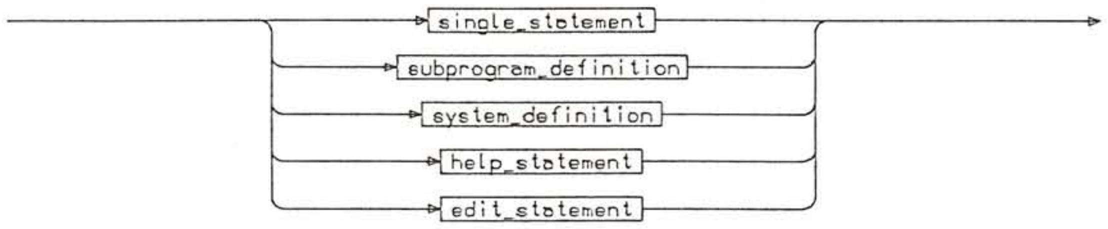
statements



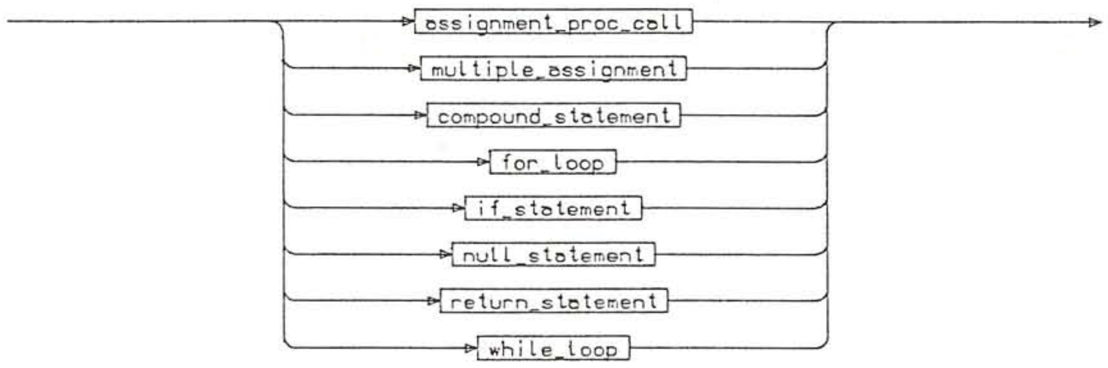
statement_terminator



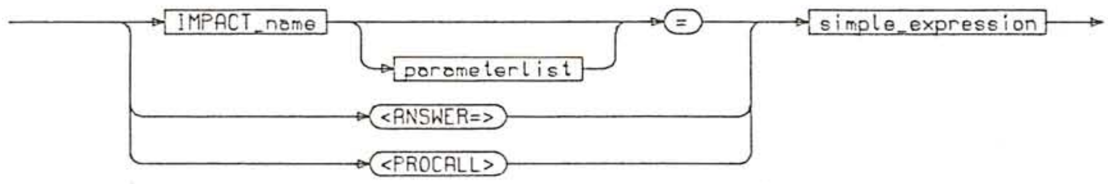
global_statement



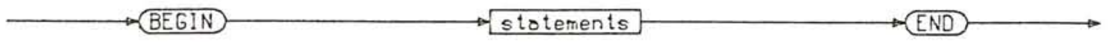
single_statement



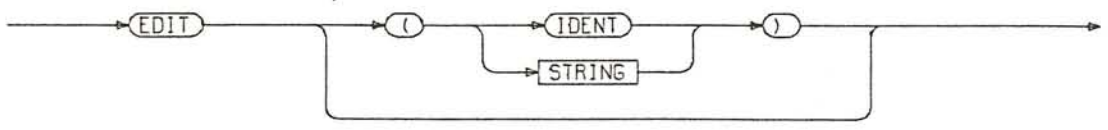
assignment_proc_call



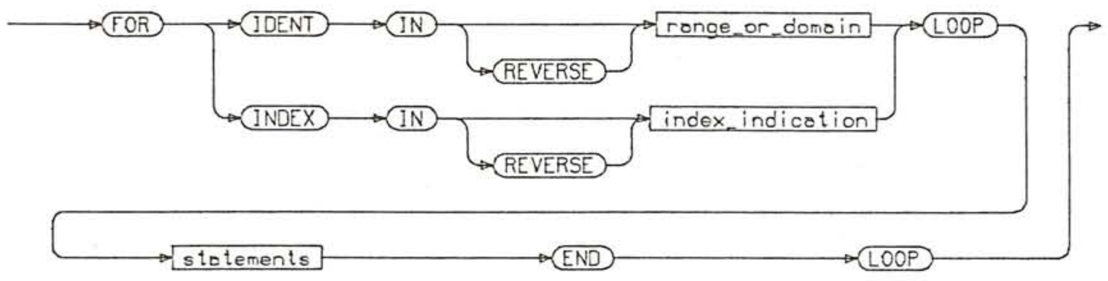
compound_statement



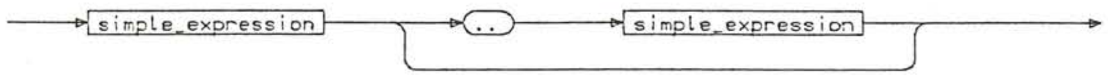
edit_statement



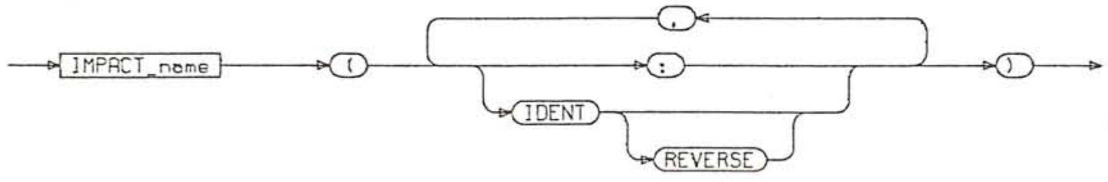
for_loop



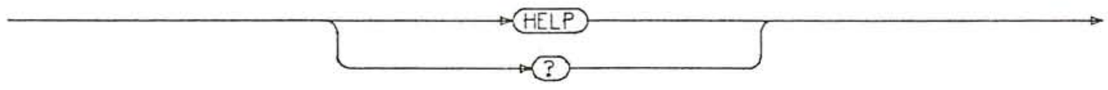
range_or_domain



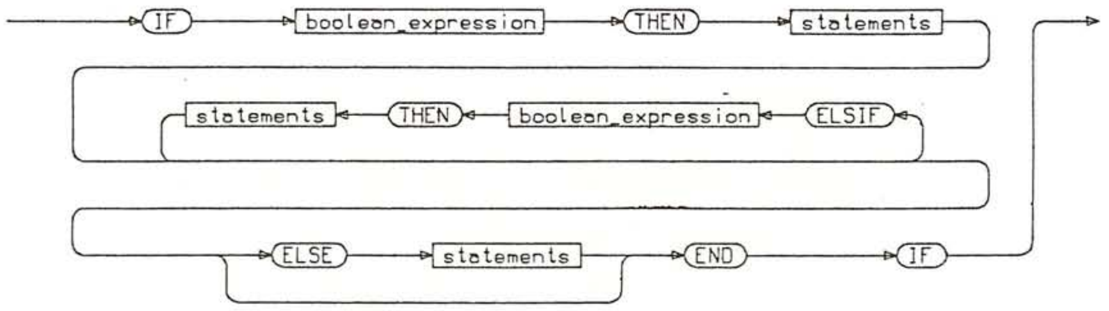
index_indication



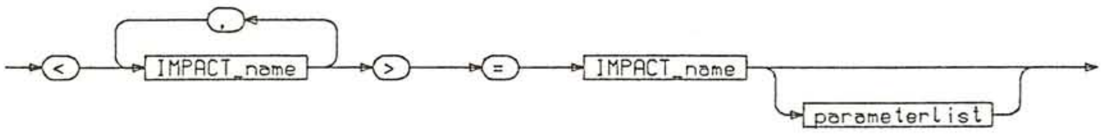
help_statement



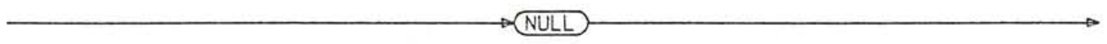
if_statement



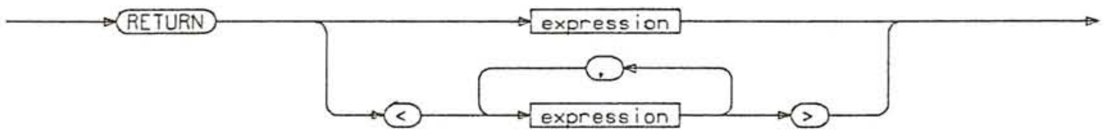
multiple_assignment



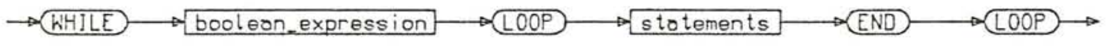
null_statement



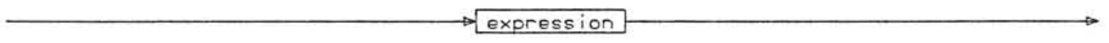
return_statement



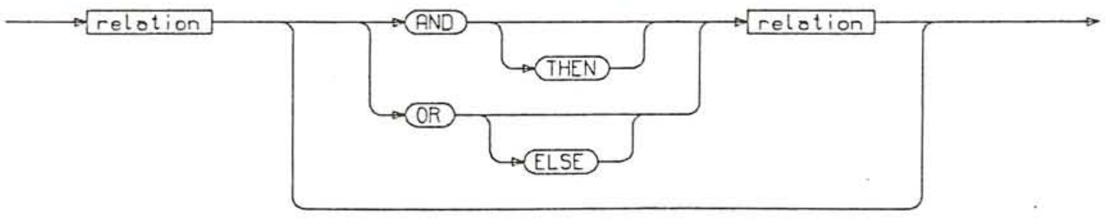
while_loop



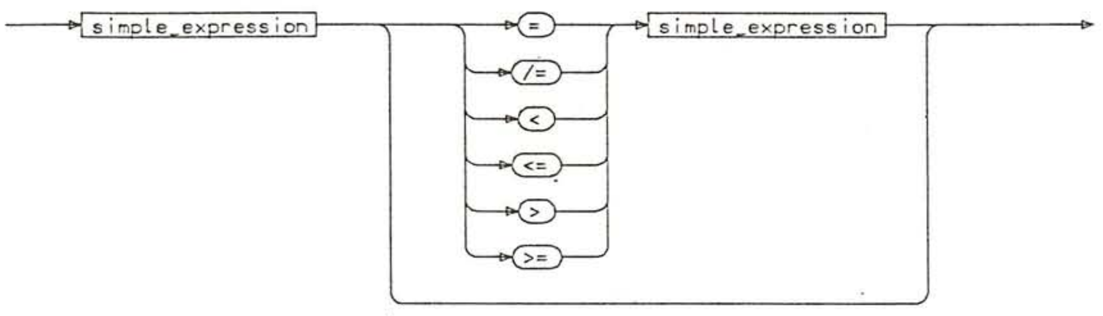
boolean_expression



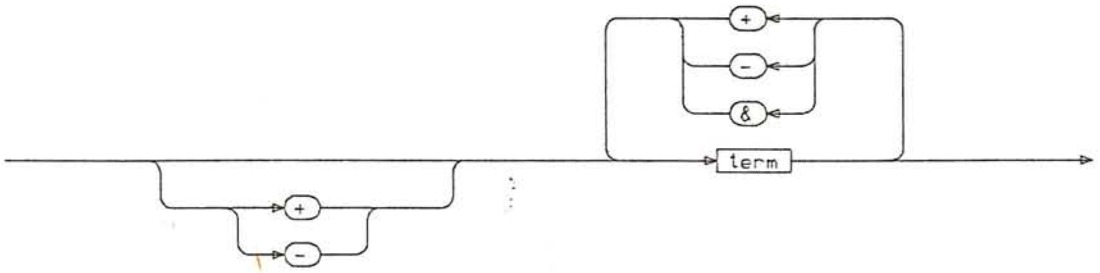
expression



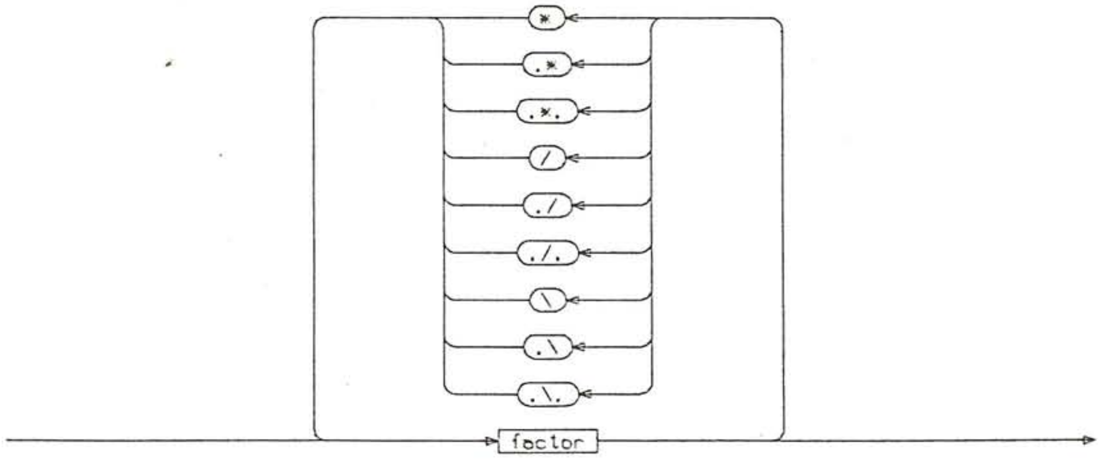
relation



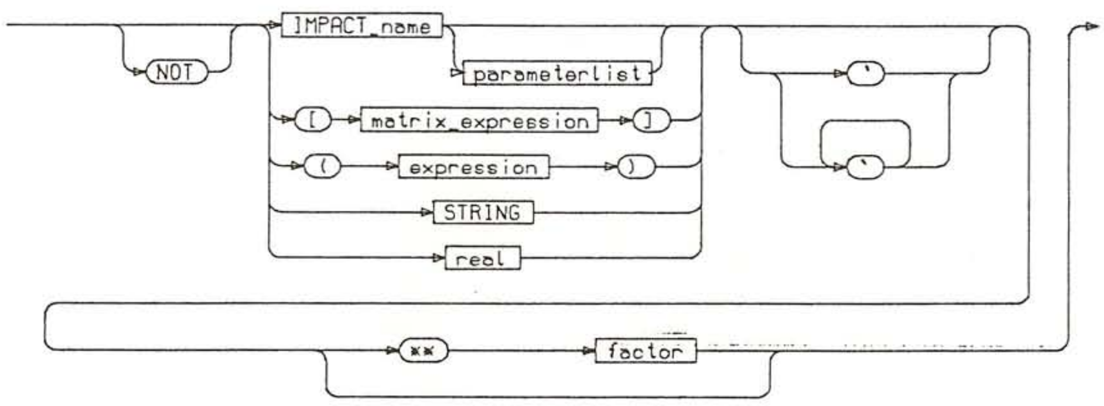
simple_expression



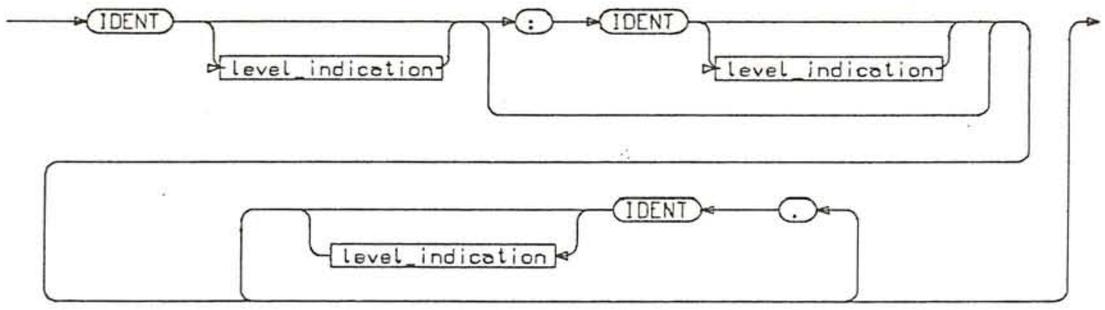
term



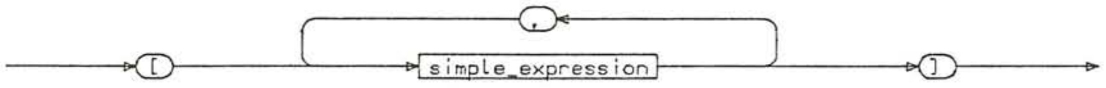
factor



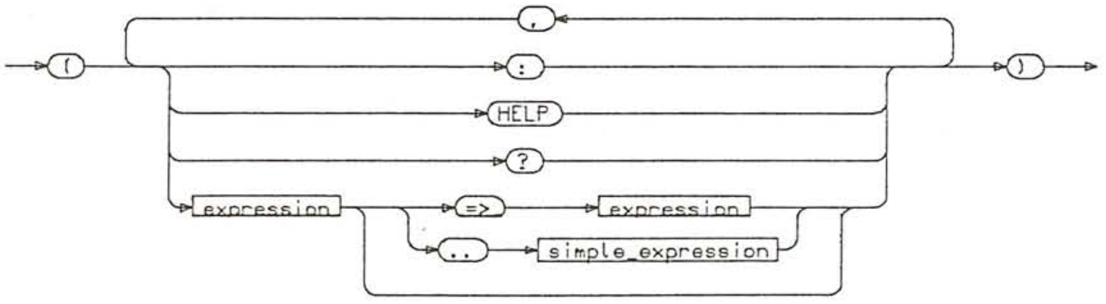
IMPACT_name



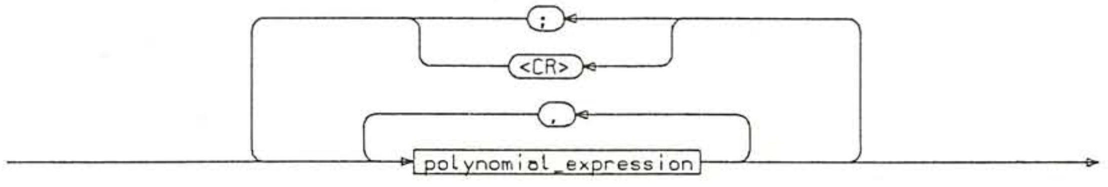
level_indication



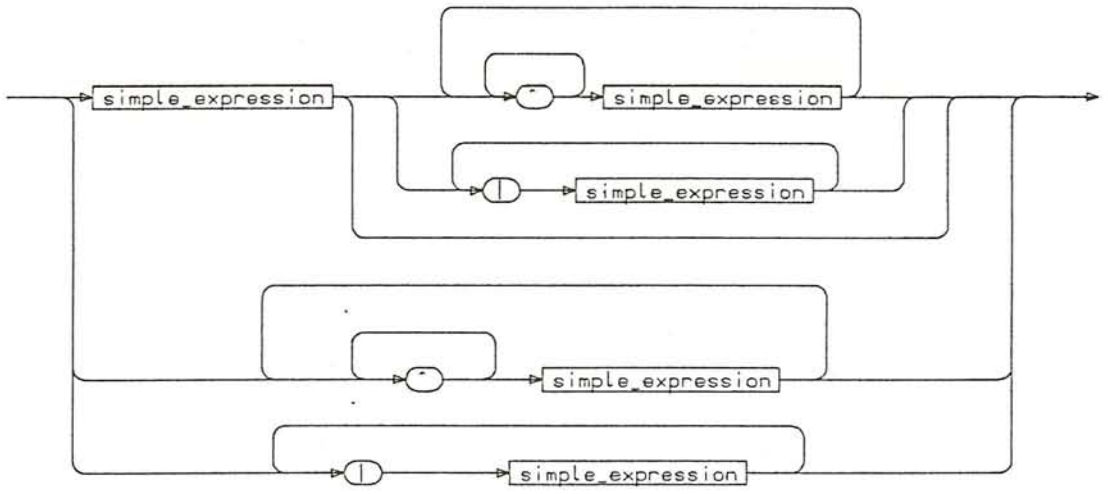
parameterlist



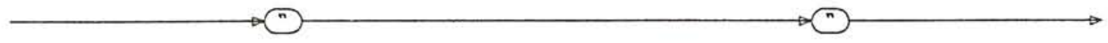
matrix_expression



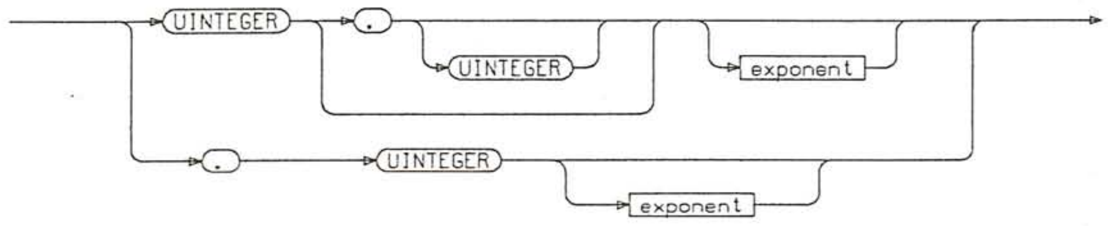
polynomial_expression



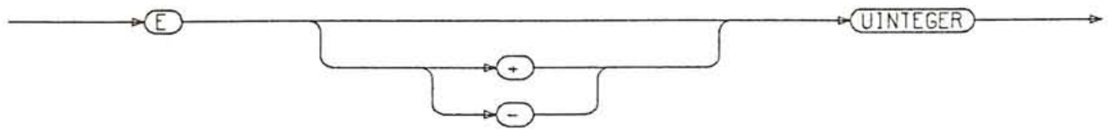
STRING



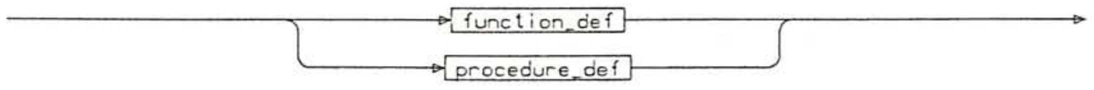
real



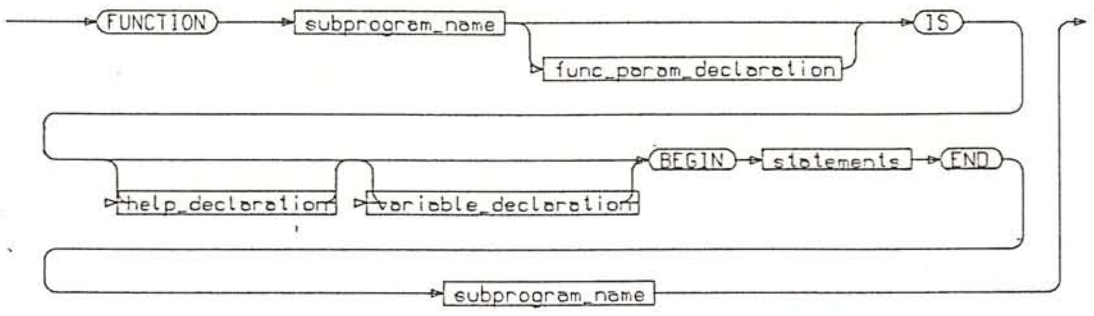
exponent



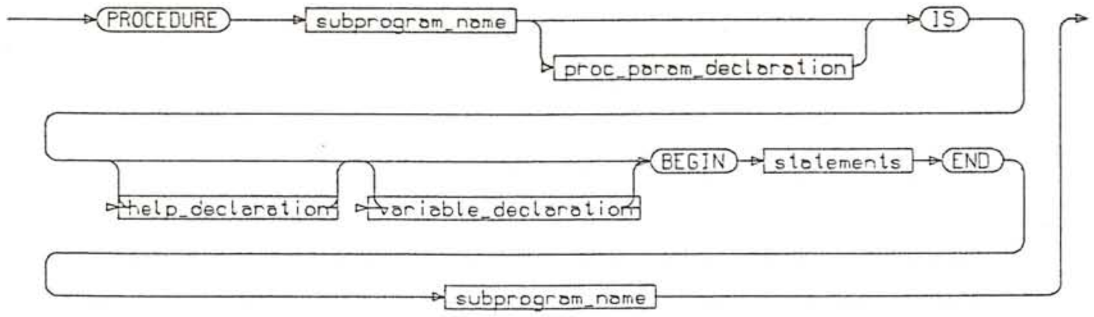
subprogram_definition



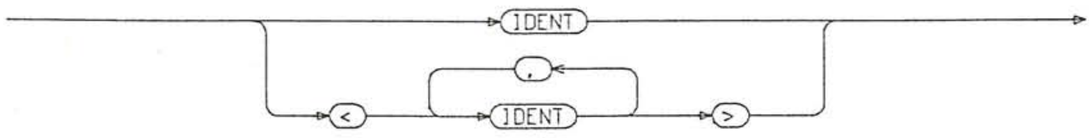
function_def



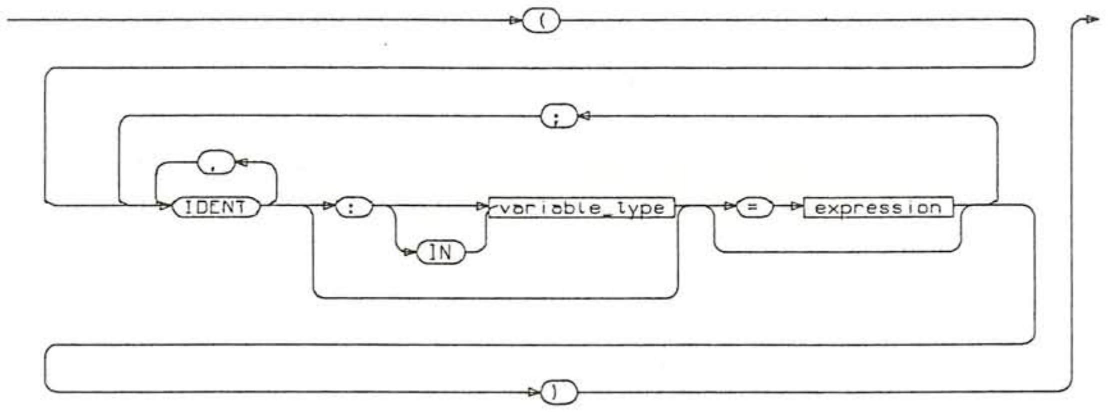
procedure_def



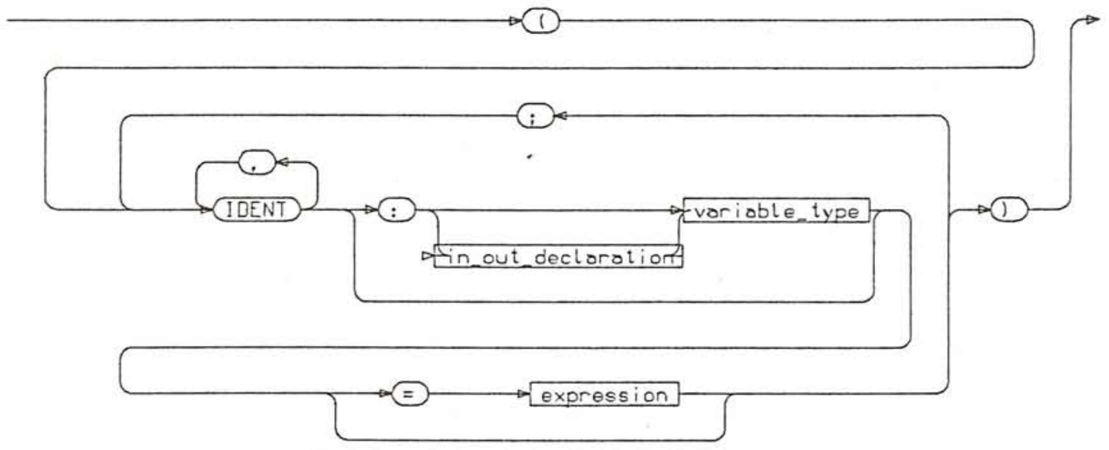
subprogram_name



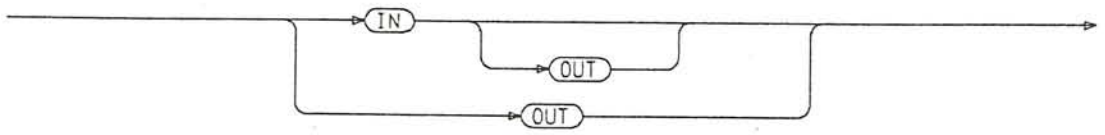
func_param_declaration



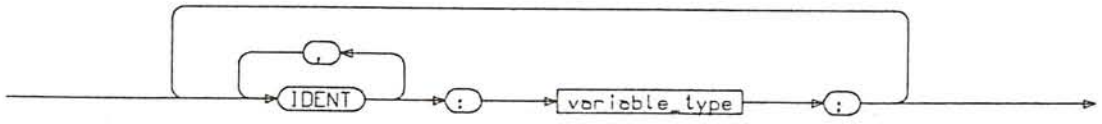
proc_param_declaration



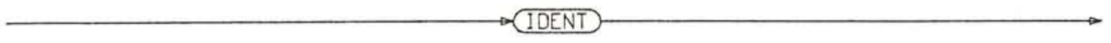
in_out_declaration



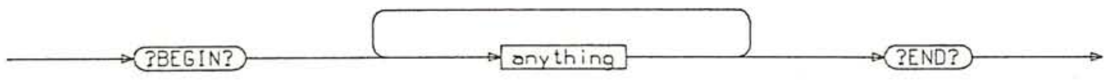
variable_declaration



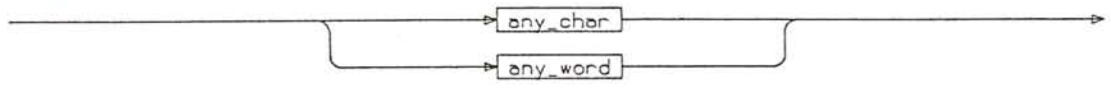
variable_type



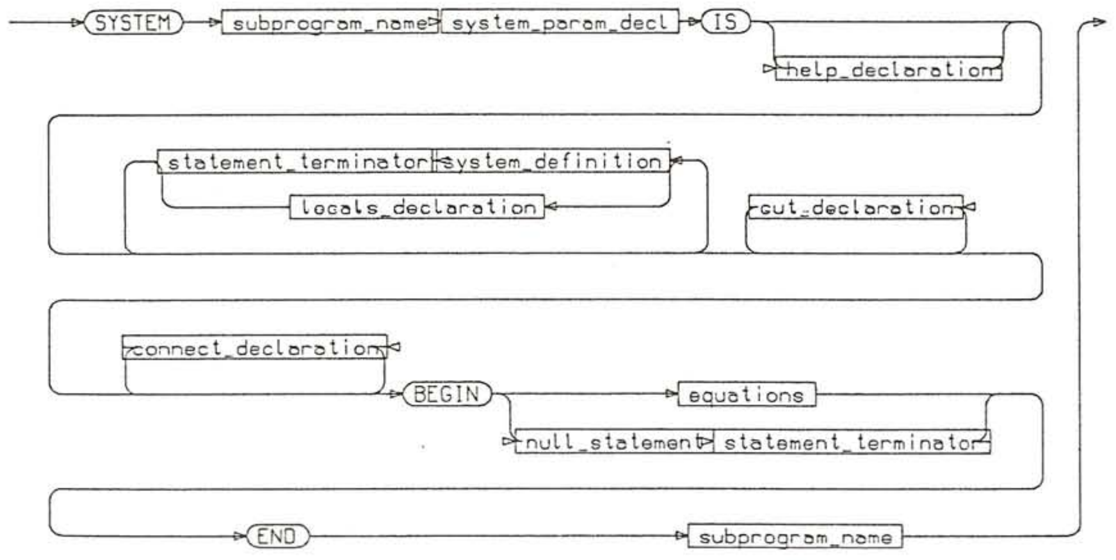
help_declaration



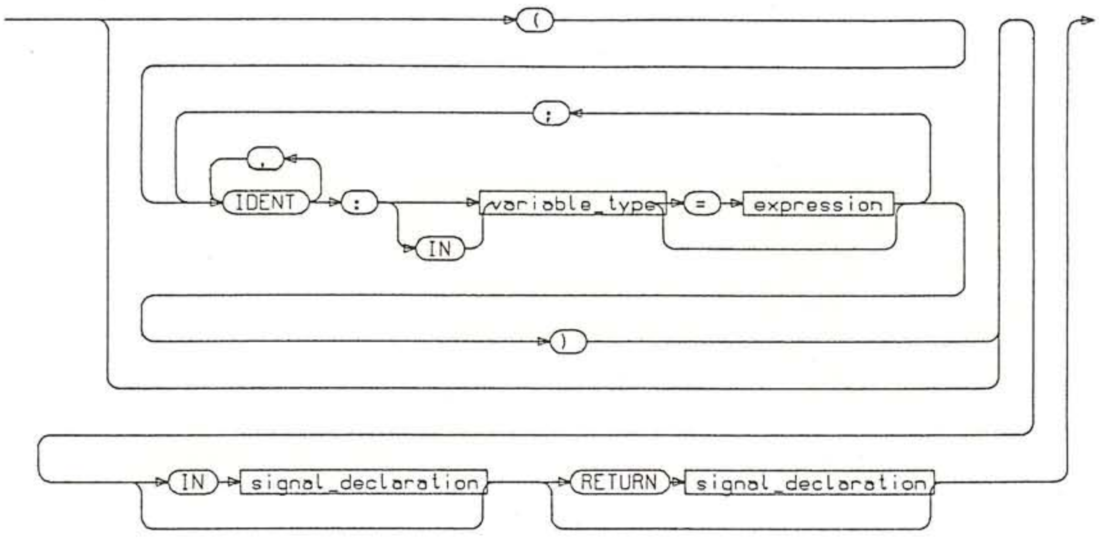
anything



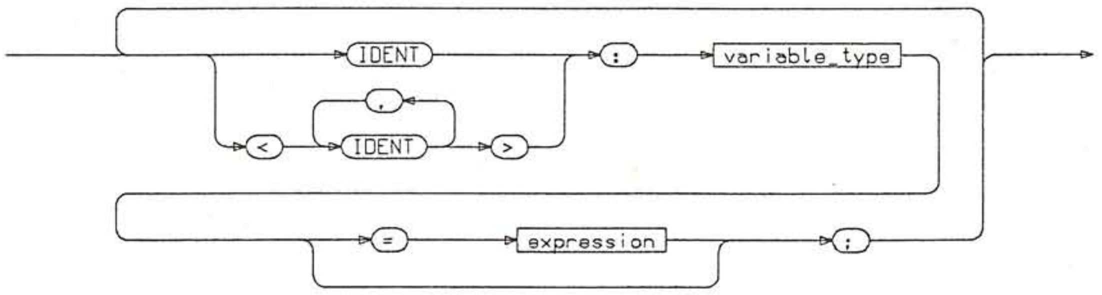
system_definition



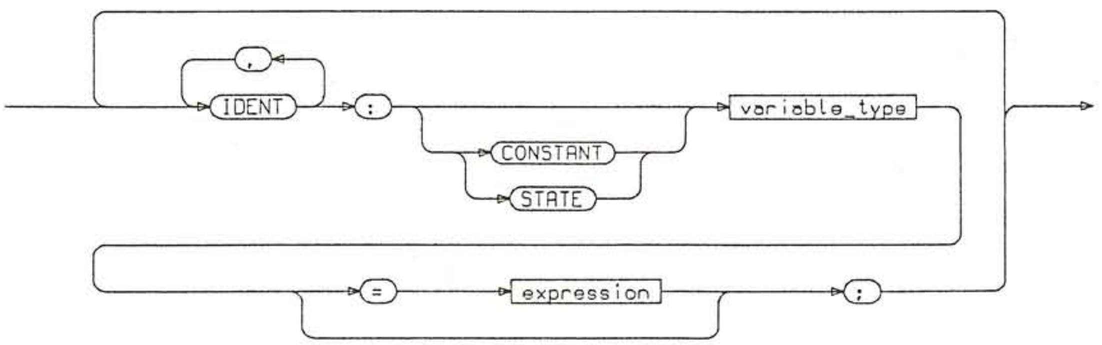
system_param_decl



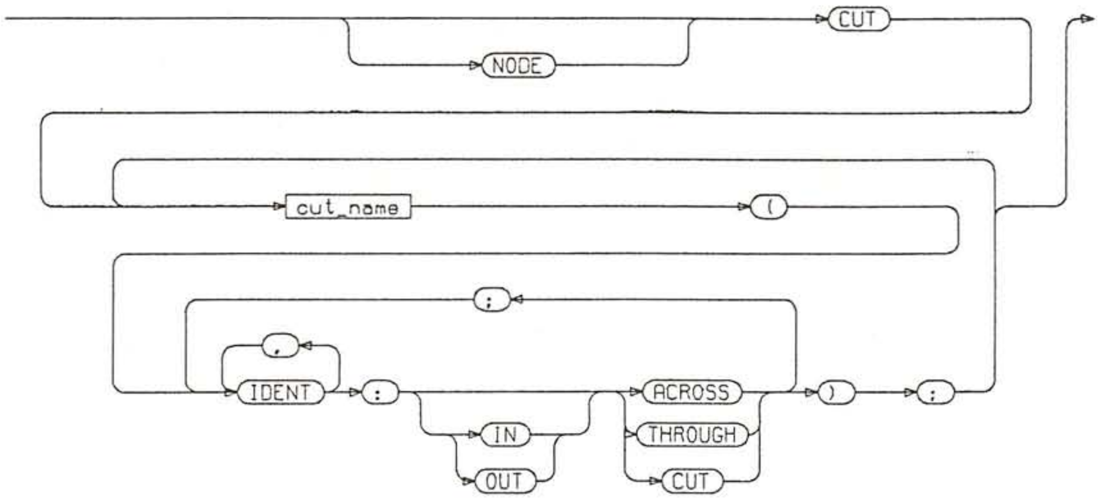
signal_declaration



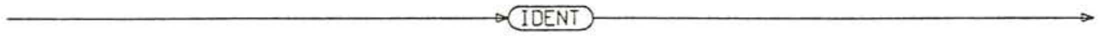
locals_declaration



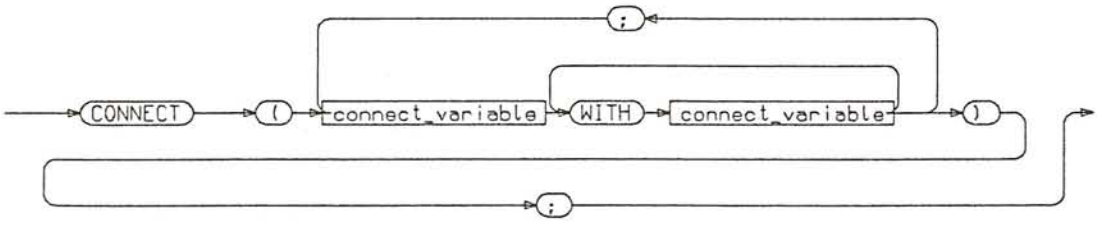
cut_declaration



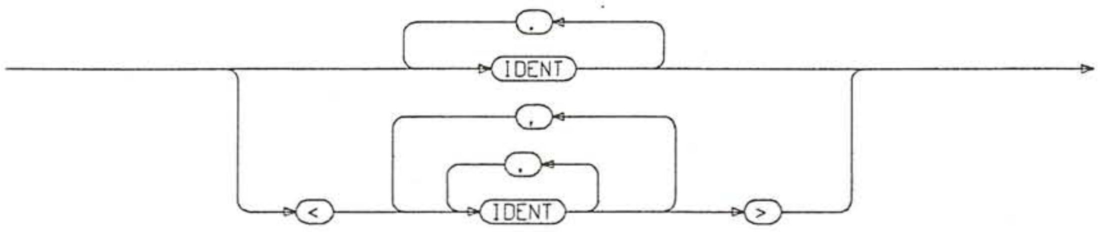
cut_name



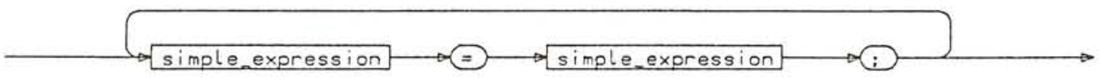
connect_declaration



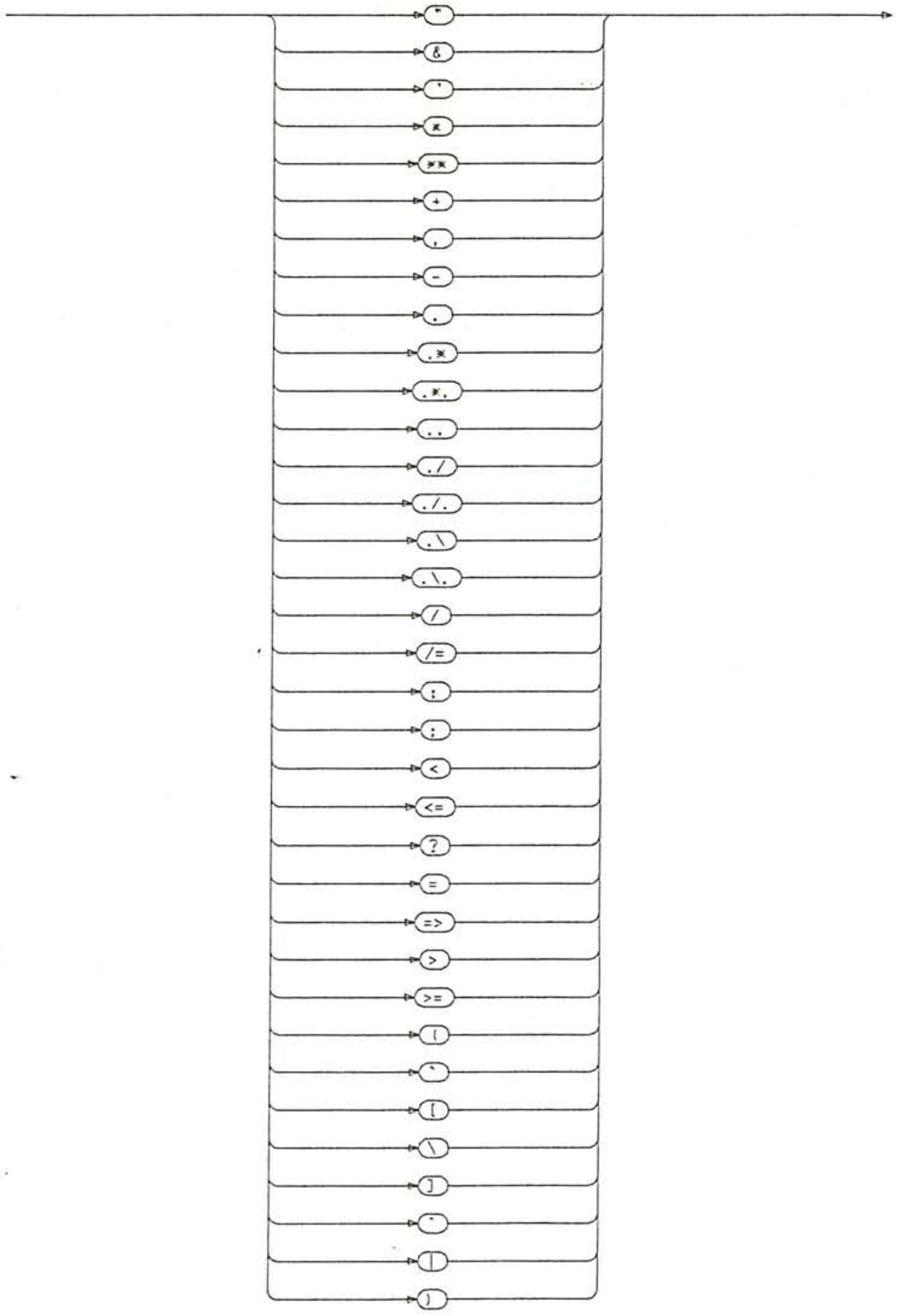
connect_variable



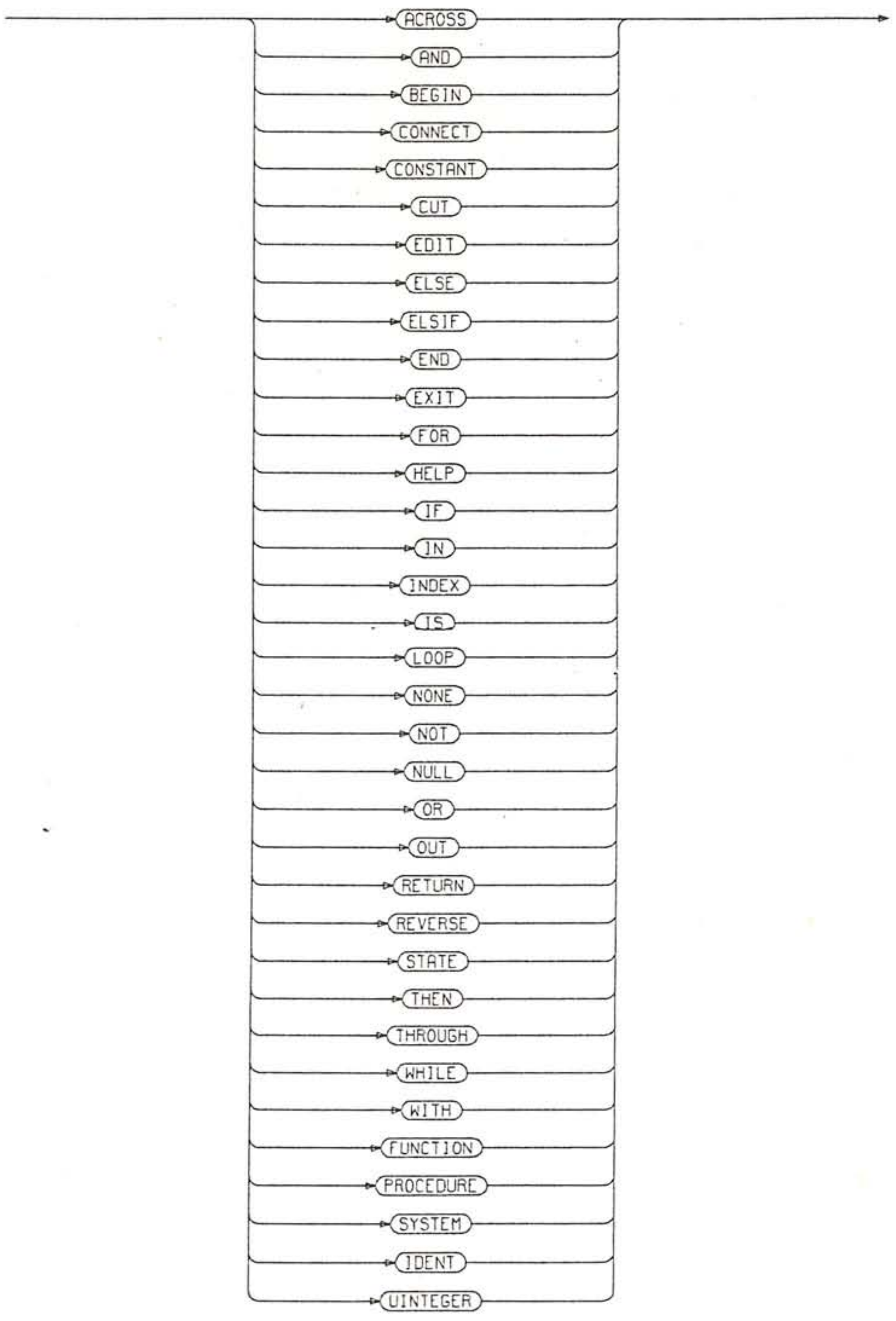
equations



any_char



any_word


ACROSS
AND
BEGIN
CONNECT
CONSTANT
CUT
EDIT
ELSE
ELSIF
END
EXIT
FOR
HELP
IF
IN
INDEX
IS
LOOP
NONE
NOT
NULL
OR
OUT
RETURN
REVERSE
STATE
THEN
THROUGH
WHILE
WITH
FUNCTION
PROCEDURE
SYSTEM
IDENT
UINTEGER

Appendix B

LIST OF PUBLICATIONS

F.E. Cellier, M. Rimvall, and A.P. Bongulielmi. Discrete processes in COSY. In F. Maceri, editor, *Procedings First European Simulation Meeting on Simulation Methodology*, Cosenza, Italy, April 9–11 1981.

M. Rimvall and F.E. Cellier. GASP-VI: Ein Simulationspaket für gemischt kontinuierliche und diskrete prozess-orientierte Simulation. In M. Goller, editor, *Proceeding ASIM-82*, pages 155–165, Informatik Fachberichte, Volume 56, Springer Verlag, Berlin, April 26–28 1982.

M. Rimvall and F.E. Cellier. The GASP-VI simulation package for process-oriented combined continuous and discrete system simulation. In W.F. Ames, editor, *Proceeding 10th IMACS World Congress on Simulation and Scientific Computation*, pages 413-416, Volume 1, Montreal, Canada, August 8–13 1982.

F.E. Cellier and M. Rimvall. Computer aided control systems design. In W. Ameling, editor, *Proc. First European Simulation Conference ESC'83*, Informatik Fachberichte, Volume 71, Springer Verlag, Berlin, 1983.

M. Rimvall. On the use of Ada in CACSD. In *Proc. Workshop on Computer Aided Control Systems Design*, pages 7–11, Institute of Measurement and Control, Brighton, England, 19–21 September 1984.

M. Rimvall and F.E Cellier. IMPACT - Interactive Mathematical Program for Automatic Control Theory. In A. Bensoussan and J.L. Lions, editors, *Proc. 6'th International Conference on Analysis and Optimization of Systems*, pages 578–597, Springer Verlag, Berlin, 1984. Lecture notes in Control and Information Sciences, volume 63.

M. Rimvall and F.E. Cellier. MIDGET - Ein flexibles, simulationstechnisches Entwicklungssystem. In F. Breitenecker and W. Kleinert, editors, *Proceeding 2. Symposium Simulationstechnik, ASIM-84*, pages 470–474, Informatik Fachberichte, Volume 85, Springer Verlag, Berlin, September 25–27 1984.

M. Mansour, W. Schaufelberger, F.C. Cellier, G. Maier, and M. Rimvall. The use of computers in the education of control engineers at ETH Zurich. *European Journal of Engineering Education*, 9:135–151, 1984.

M. Rimvall and L. Bomholt. A flexible man-machine interface for CACSD applications. In *Preprints 3rd IFAC/IFIP international symposium on Computer Aided Design in Control and Engineering Systems*, Copnehagen, Denmark, July-August 1985.

M. Rimvall and F.E. Cellier. The matrix environment as enhancement to modeling and simulation. In *Proceeding 11th IMACS World Congress on Simulation and Scientific Computation*, Oslo, Norway, August 5–9 1985.

M. Rimvall and F.E. Cellier. A structural approach to CACSD. In M. Jamshidi and C.J. Herget, editors, *Advances in Computer-Aided Control Systems Engineering*, pages 149–158, North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

M. Rimvall, M. Mansour, and W. Schaufelberger. Computer-aided control-systems design in undergraduate education at ETH Zurich. *Transaction Institute Measurement and Control*, 7:90–96, 1985.

M. Rimvall, D. Slagstad, and T. Iversen. Computer aided modeling and simulation using a direct-executing simulation language. In *Proceeding 11th IMACS World Congress on Simulation and Scientific Computation*, Oslo, Norway, August 5–9 1985.

F.E. Cellier and M. Rimvall. Distributed modelling and data base management in simulation. In *Proceeding SCS Multiconference*, San Diego, CA, January 24-26 1985.

M. Mansour, M. Rimvall, and W. Schaufelberger. Computer aided design of control systems - an integrated approach. In *Preprints 3rd IFAC/IFIP international symposium on Computer Aided Design in Control and Engineering Systems*, pages 28–33, Copenhagen, Denmark, 1985.

M. Rimvall. CACSD software and man-machine interfaces of modern control environments (keynote address). In *Proc. Workshop on Computer Aided Control Systems Design*, pages 21–28, Institute of Measurement and Control, Salford, England, 2–4 July 1986.

M. Rimvall and F.E. Cellier. Evolution and perspectives of simulation languages following the CSSL standard. *Modeling, identification and control*, 9:181–199, 1986.

M. Rimvall, F. Schmid, and F.E. Cellier. The different modeling capabilites of IMPACT. In *Proceedings IEEE Control Systems Society 3rd Symposium on Computer-Aided Control System Design (CACSD)*, Arlington, VA, September 24–26 1986.

F.E. Cellier and M. Rimvall. Computer-aided control systems design - techniques and tools. In N. Kheir, editor, *Systems Modeling and Computer Simulation*, Marcel Dekker, NY, 1986, To Appear.

A. Fischlin, M. Mansour, M. Rimvall, and W. Schaufelberger. Simulation and computer aided control system design in engineering education. In *Proceedings of the IFAC/IMACS International Symposium on Simulation of Control Systems*, Vienna, Austria, September 22-26 1986.

Bibliography

ACM/SIGGRAPH. Status report of the graphics standards committee. *Computer Graphics*, 13(3), 1979.

(ACSL) Advanced Continuous Simulation Language, Reference Manual. Mitchell and Gauthier Associates, Concord, MA 01742, 4th edition, 1986.

AdaIC List of Validated Ada Compilers. August 1986. Ada Information Clearinghouse Newsletter (AdaIC), 3D139 (1211 Fern., C107), The Pentagon, Washington, DC 20301.

P. Agathoklis, F. C. Cellier, M. Djordjevic, P. O. Grepper, and F. J. Kraus. Educational aspects of using computer-aided design in automatic control. In M. A. Cuenod, editor, *IFAC Symposium on Computer-Aided Design of Control Systems*, pages 441–446, Pergamon Press, London, 1979.

P. Agatoklis. 1986. Private communication.

A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

U. Amman. *Error Recovery in recursive descent parsers. Runtime Storage Organization.* Technical Report 25, Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1978.

ANSI/MIL-STD 1815 A, Reference manual for the Ada programming language. U.S. Department of Defense, 1983.

ANSI X3.124-1985, Functional description of GKS. ANSI, New York, 1985.

K.J. Åström. Computer aided tools for control system design. In M. Jamshidi and C.J. Herget, editors, *Advances in Computer-Aided Control Systems Engineering*, pages 3–40, North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

K.J. Åström. Computer-aided analysis and design of control systems - a perspective. *IEEE Control Systems Magazine*, 3(2):4–16, May 1983.

K.J. Åström. Computer-aided design of control systems. In A. Bensoussan and J.L. Lions, editors, *Proc. 6'th International Conference on Analysis and Optimization of Systems*, pages 549–563, Springer Verlag, Berlin, 1984. Lecture notes in Control and Information Sciences, volume 63.

K.J. Åström. *A Simnon Tutorial.* Technical Report CODEN: LUTFD2/(TFRT–3168)/1–52/(1982), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, October 1982.

D.P. Atherton, O.P. McNamara, M.D. Wadey, and A. Goucem. SUNS: the Sussex university nonlinear control systems software. In *Preprints 3rd IFAC/IFIP international symposium on Computer Aided Design in Control and Engineering Systems*, pages 173–178, Copenhagen, Denmark, 1985.

D.C. Augustin, J.C. Strauss, M.S. Fineberg, B.B. Johnson, R.N. Linebarger, and F.J. Sansom. The SCi continuous system simulation language (CSSL). *Simulation*, 9(6):281–303, December 1967.

N.J.C. Baker and P.J. Smart. The SYSMOD simulation language. In W. Ameling, editor, *Proc. First European Simulation Conference ESC'83*, pages 281–286, Informatik Fachberichte, Volume 71, Springer Verlag, Berlin, 1983.

A. Barr and E.A. Feigenbaum, editors. *The Handbook of Artificial Intelligence, Volume 1(3).* Pitman, London, 1981.

G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Studentlitteratur, Lund, Sweden, 2nd edition, 1973.

A.P. Bongulielmi and F.E. Cellier. On the usefulness of using deterministic grammars for simulation languages. *Simuletter*, 15(1):14–36, January 1984.

W. Bucher. 1984. Private communication.

CASCADE - Issues in the Design of a Computer-Aided Systems and Control Analysis and Design Environment. Oak Ridge National Laboratory, TN 37831, 1984, ORNL-TM9038. Contributions from M.T. Athens, R. Strunce, S.A. Bly, C.J. Herget, A.L. Laub, J.D. Birdwell (Project Manager), R. Cockett, R.W. Heller, R.W. Rochelle, M.T. Heath and J.P. Stovall.

CC - Program CC User's Guide. Peter M. Thompson, Systems Technology, Inc., 13766 So. Hawthorne Blvd., Hawthorne, CA 90250, 1985.

F.E. Cellier. *Combined Continuous/Discrete System Simulation by Use of Digital Computers: Techniques and Tools.* PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1979. Number ETH 6483.

F.E. Cellier. Enhanced run-time experiments for continuous system simulation languages. In F.E. Cellier, editor, *Languages for Continuous System Simulation, Proceeding SCS Multiconference*, pages 78–83, San Diego, CA, January 1986.

CINEMA. Systems Modeling Corp., P.O. Box 10074, State College, PA 16805, 1986.

W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin, 2nd edition, 1984.

The CSSL-IV Simulation Language, Reference Manual. Simulation Services Div., Nilsen Associates, 20926 Germain Street, Chatsworth, CA 91311, 1984.

CTRL-C User's Guide. Systems Control Technology, Inc., 1801 Page Mill Road, Palo Alto, CA 94304, 4.0 edition, September 1986.

M.A. Cuenod, editor. *Preprints IFAC Symposium on Computer Aided Design of Control Systems*, Pergamon Press, Oxford, 1979.

H. de Swaan Arons. Expert systems in the simulation domain. *Mathematics and Computers in Simulation*, 25:10–16, 1983.

R.A. DeCarlo and R. Saeks. *Interconnected Dynamical Systems.* Marcel Dekker, NY, 1981.

H. Domeisen, B. Dorn, T. Sultzer, and S. Studer. Interactive graphic design of signal flow diagrams for simulation. In *Preprints 3rd IFAC/IFIP international symposium on Computer Aided Design in Control and Engineering Systems*, pages 241–245, Copenhagen, Denmark, 1985.

J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart. *LINPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, 1979.

EAGLES/Controls Documentation. Lawrence Livermore National Laboratory, P.O.Box 808, Livermore, CA 94550, May 1986.

ELCS - The Extended List of Control Software, Number 2. D. Frederick, C. Herget, R. Kool and M. Rimvall, editors, Request copy from M. Rimvall, Department of Automatic Control, ETH-Zentrum, CH-8092 Zürich, Switzerland, June 1986.

H. Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978. Number CODEN: LUTFD2/(TFRT-1015)/1–226/(1978).

H. Elmqvist and S.E. Mattsson. A simulator for dynamic systems using graphics and equations for modelling. In *Proceedings IEEE Control Systems Society 3rd Symposium on Computer-Aided Control System Design (CACSD)*, Arlington, VA, September 24–26 1986.

T. Essebo. *Machine Code Generation for Simnon on VAX-11.* Technical Report CODEN: LUTFD2/(TFRT-7217)/1–045/(1981), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1981.

D.K. Frederick, T. Sadeghi, and R.P. Kraft. Computer-aided control system analysis and design using interactive computer graphics. In M. Jamshidi and C.J. Herget, editors, *Advances in Computer-Aided Control Systems Engineering*, pages 265–275, North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

B.S Garbow, J.M. Boyle, J.J. Dongarra, and C.B. Moler. *Matrix Eigensystem Routines, EISPACK Guide Extensions.* Volume 51 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, 1977.

J.B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.

S.D. Goodfellow and N. Munro. An integrated environment for computer aided control systems engineering. In *Preprints 3rd IFAC/IFIP international symposium on Computer Aided Design in Control and Engineering Systems*, pages 104–109, Copenhagen, Denmark, 1985.

P. O. Grepper and M. Djordjevic. A computer program for interactive control system design. In *Preprints IFAC Symposium on Trends in Automatic Control Education*, Barcelona, Spain, 1977.

N.E. Hansen and P.M. Larsen, editors. *Preprints 3rd IFAC-IFIP international symposium on Computer Aided Design in Control and Engineering Systems (CADCE'85)*, Copenhagen, Denmark, 1985.

A. Hearn. *REDUCE 2 User's Manual.* Rep. UCP-19, University of Utah, Salt Lake City, 1973.

C.J. Herget and A.J. Laub. Special issue on computer-aided control system design programs. *IEEE Control Systems Magazine*, 2(4), December 1982.

C.J. Herget and A.J. Laub. Special section on computer-aided design of control systems: systems and algorithms. *Proceedings of the IEEE*, 72(12), December 1984.

A. Hindmarsh. ODEPACK: a systematized collection of ODE solvers. In R.S. Stepleman, editor, *Numerical Methods for Scientific Computing*, pages 55–64, North-Holland, Elsevier Science Publishers, Amsterdam, 1983.

IBM. *System/360 Scientific Subroutine Package, Version III Programmers Manual.* IBM, 1968.

Proceedings of the IEEE Control Systems Society 1st Symposium on Computer-Aided Control System Design, MIT, Cambridge, MA, USA, September 28–30 1983. Abstracts.

Proceedings of the IEEE Control Systems Society 2nd Symposium on Computer-Aided Control System Design (CACSD), Santa Barbara, MA, USA, March 13–15 1985. Abstracts.

Proceedings of the IEEE Control Systems Society 3rd Symposium on Computer-Aided Control System Design (CACSD), Arlington, VA, September 24–26 1986.

Minutes from the first meeting of the IFAC Working Group on Guidelines for CACSD software, Arlington, VA. September 24 1986. To be ordered from M. Rimvall, Dept. of Automatic Control, Swiss Federal Institute of Technology, Zürich, Switzerland.

IMSL Library, User's Manual. IMSL, NBC Building, 2500 ParkWest Tower One, 2500 CityWest Boulevard, Houston, TX 77042, USA, 9.2 edition, November 1982.

J.R. James, D.K. Frederick, and J.H. Taylor. The use of expert-system programming techniques for the design of lead-lag compensators. In *IEE International Conference Control 85*, pages 180–185, 1985.

M. Jamshidi and C.J. Herget, editors. *Computer Aided Control Systems Engineering.* North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

K. Jensen and N. Wirth. *Pascal, User Manual and Report.* Springer Verlag, Berlin, 2nd edition, 1975.

J. Joss. *Algorithmisches Differenzieren.* PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1976. Number ETH 5757.

T. Kailath. *Linear Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

B.W. Kernighan and D.M. Ritchie. *The C Programming Lanugage.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

R.A. King and J.O Gray. A graphical man-machine interface for CAD and simulation of dynamic system. In *Proc. 6th European Conference on Electrotechnics, Computers in Communication and Control (EUROCON 84)*, Brighton, England, September 26–28 1984.

G.A. Korn. EARLY DESIRE. *Mathematics and Computers in Simulation*, 24(1):30–36, February 1982.

J.E. Larsson and P. Persson. Knowledge representation by scripts in an expert interface. In *Proceedings American Control Conference, ACC'86*, pages 1159–1162, Seattle, WA, June 18–20 1986.

L.L. Lehman, S. Houtchens, M. Navab, and S.C. Shah. Automated synthesis of Ada real time control software. In *Proceedings IEEE Control Systems Society 3rd Symposium on Computer-Aided Control System Design (CACSD)*, Arlington, VA, September 24–26 1986.

G.G. Leininger, editor. *Computer Aided Design of Multivariable Technological Systems – Proceedings 2rd IFAC Symposium, West Lafayette, IN, 15–17 September 1982*, Pergamon Press, Oxford, 1982.

J. Lewi, K. De Vlaminick, J. Huens, and M. Huybrechts. *Project LILA - The ELL(1) Generator Error Recovery.* Technical Report CW8, Applied Mathematics and Programming Division, Katholieke Universiteit Leuven, Belgium, September 1976.

J. Lewi, K. De Vlaminick, J. Huens, and E. Steegmans. *Project LILA - Error Recovery in Batch ELL(1) Parsers: Automatic Generation.* Technical Report CW30, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium, January 1983.

J.N. Little, A. Emami-Naeini, and S.N. Bangert. CTRL-C and matrix environments for the computer-aided design of control systems. In A. Bensoussan and J.L. Lions, editors, *Proc. 6'th*

International Conference on Analysis and Optimization of Systems, pages 191–205, Springer Verlag, Berlin, 1984. Lecture notes in Control and Information Sciences, volume 63.

R.G. Loeliger. *Threaded interpretative languages: their design and implementation.* Byte Books, Petersborough, NH, 1981.

J.M. Maciejowski. *A Core Data Model for Computer-Aided Control Engineering.* Technical Report CUED/F-CAMS/TR.257, Cambridge University, Engineering Department, December 1985.

J.M. Maciejowski. Data structures for control systems design. In *Proc. 6th European Conference on Electrotechnics, Computers in Communication and Control (EUROCON 84)*, Brighton, England, September 26–28 1984.

MACSYMA. Symbolics, Inc., Four Cambridge Center, Cambridge, MA 02143, 1986.

G. Maier. *Entwurf und Realisierung einer Methode zur Exceptionbehandlung und Synchronisation in Echtzeitprogrammen.* PhD thesis, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1979. Number ETH 7583.

M. Mansour, M. Rimvall, and W. Schaufelberger. Computer aided design of control systems - an integrated approach. In *Preprints 3rd IFAC/IFIP international symposium on Computer Aided Design in Control and Engineering Systems*, pages 28–33, Copenhagen, Denmark, 1985.

J.M. Mason, C.P. Neuman, and B.H. Krogh. CACHE: an interactive control system analysis and design package. *IEEE Transactions on Education*, 28(3):143–149, August 1985.

C. Moler. *MATLAB, User's Guide.* Department of Computer Science, University of New Mexico, Albuquerque, USA, 1980.

C. Moler, J. Little, S. Bangert, and S. Kleinman. *PC-Matlab, User's Guide.* The MathWorks, Inc., 158 Woodland St., Sherborn, MA 01770, U.S.A, 2.0 edition, November 1985.

J.J. Moré, B.S. Garbow, and K.E. Hillstrom. *User Guide for MINPACK-1.* Technical Report ANL-80-74, Argonne National Laboratory, 1980.

N. Munro. The UMIST control system design and synthesis suites. In M. A. Cuenod, editor, *IFAC Symposium on Computer-Aided Design of Control Systems*, pages 343–348, Pergamon Press, London, 1979.

D.A. Poplawski. *Error Recovery for Extended LL-Regular Parsers.* PhD thesis, Purdue University, West Lafayette, IN, 1978.

J.E. Potter. Matrix quadratic solutions. *SIAM Journal of Applied Mathematics*, 14(3):496–501, May 1966.

L.B. Rall. *Automatic differentiation: Techniques and applications.* Springer Verlag, Berlin, 1981.

J.R. Rice and R.F. Boisvert. *Solving elliptic problems using ELLPACK.* Springer Verlag, Berlin, 1985.

M. Rimvall. *Ada-Guidelines for the IMPACT Project.* Technical Report, Dept. of Automatic Control, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, July 1986.

M. Rimvall. *IMPACT, Interactive Mathematical Program for Automatic Control Theory, a Preliminary User's Manual.* Department of Automatic Control, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, 1983.

S. Roski. DOD-STD-2167A, Appendix D. Ada design and coding standards. *ACM Ada Letters*, 6(5):36–44, September, October 1986. Draft version.

W. Schaufelberger, H. Good, and A Itten. Education for microcomputer applications in control. In *IFAC symposium on Microcomputer Application in Process Control*, Istambul, July 1986.

C. Schmid. KEDDC - a computer-aided analysis and design package for control systems. In M. Jamshidi and C.J. Herget, editors, *Advances in Computer-Aided Control Systems Engineering*, pages 159–180, North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

C. Schmid. A workstation concept for computer aided analysis and design of control systems. In *Preprints IFAC-IFIP-IMACS 7th Conference on Digital Computer Applications to Process Control*, pages 691–694, Vienna, Austria, 1985.

S.C. Shah, M.A. Floyd, and L.L. Lehman. MATRIX$_X$: control design and model building CAE capabilities. In M. Jamshidi and C.J. Herget, editors, *Computer Aided Control Systems Engineering*, pages 181–207, North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

B.T. Smith, J.M. Boyle, J.J Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema, and C.B. Moler. *Matrix Eigensystem Routines, EISPACK Guide.* Volume 6 of *Lecture Notes in Computer Science*, Springer Verlag, Berlin, 1974.

I. Sommerville. *Software Engineering.* Addison-Wesley, Reading, MA, 2nd edition, 1985.

H.A. Spang. The federated computer-aided control design system. In M. Jamshidi and C.J. Herget, editors, *Advances in Computer-Aided Control Systems Engineering*, pages 209–227, North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

SYSMOD User Manual. Systems Designers, Farnborough, England, 1.0 edition, April 1986.

J.H. Taylor and D.K. Frederick. An expert system architecture for computer-aided control engineering. *Proceedings of the IEEE*, 72(12):1795–1805, December 1984.

The TESS User's Manual. Pritsker and Associates, P.O. Box 2413, West Lafayette, IN 47906, USA, 2.2 edition, Juli 1986.

T.L. Trankle, P. Sheu, and U.H. Rabin. Expert systems architecture for control system desing. In *Proceedings American Control Conference, ACC'86*, pages 1163–1169, Seattle, WA, June 18–20 1986.

K.F. VanNeste. Ada coding standards and conventions. *ACM Ada Letters*, 6(1):41–48, January, February 1986.

R. Walker, C. Gregory Jr., and S. Shah. MATRIX$_X$, a data analysis, system identification, control design and simulation package. *IEEE Control Systems Magazine*, 2(4):30–37, 1982.

R.A. Walker, S.C. Shah, and N.K. Gupta. Computer-aided engineering (CAE) for system analysis. *Proceedings of the IEEE*, 72(12):1732–1745, December 1984.

P.J. West, S.P. Bingulac, and W.R. Perkins. L-A-S: a computer-aided control system design language. In M. Jamshidi and C.J. Herget, editors, *Advances in Computer-Aided Control Systems Engineering*, pages 243–261, North-Holland, Elsevier Science Publishers, Amsterdam, 1985.

WGS. *Implementation and documentation standards for the basic subroutine library SYCOT (SYstems and COntrol Tools)*. Technical Report, Werkgroep Programmatuur (Working Group on Software WGS), 1983. Order from R. Kool (secr.), HG 9.89, Eindhoven University of Technology, NL-6600MB Eindhoven, the Netherlands.

WGS. *An Inventory of Basic Software for Computer Aided Control Systems Design (CACSD)*. Technical Report, Werkgroep Programmatuur (Working Group on Software WGS), 1985. WGS-Report 85–1, Order from R. Kool (secr.), HG 9.89, Eindhoven University of Technology, NL-6600MB Eindhoven, the Netherlands.

J. Wieslander. *Interaction in Computer Aided Analysis and Design of Control Systems*. PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1979. Number CODEN: LUTFD2/(TFRT–1019)/1–222/(1979).

J. Wieslander. *Interactive Programs - General guide*. Technical Report CODEN: LUTFD2/(TFRT–3156)/1–30/(1980), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1980.

P.H. Winston and B.K.P. Horn. *LISP*. Addison-Wesley Publishing Company, Reading, MA, 1981.

N. Wirth. *Compilerbau.* Teubner Verlag, Stuttgart, FRG, 1977.

N. Wirth. *Programming in Modula-2.* Springer Verlag, Berlin, 3rd edition, 1985.

S. Wolfram. Symbolic mathematical computation. *Communications of the ACM*, 28(4):390–394, April 1985.

B.P. Zeigler, editor. *Expert Systems and Simulation Models*, The University of Arizona in cooperation with NASA and AFCEA, November 18-19 1985.

B.P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation.* John Wiley and Sons, NY, 1984.

B.P. Zeigler. *Theory of Modelling and Simulation.* John Wiley and Sons, NY, 1976.

Curriculum Vitae

I was born in Laholm, Sweden on February 24 1957. I obtained my primary level education in Laholm and Halmstad (Sweden). In 1974–75 I spent one year as an exchange student and High-School senior in Greenfield, Illinois. In the spring of 1976 I graduated from Sannarps Gymnasium (Halmstad, Sweden) with a specialization in the natural sciencis and mathematics. The same fall I commenced my studies at the Department of "Technical Physics", Lund Institute of Technology (LTH), Lund, Sweden. Between my first and second year of higher education I spent 11 months in the Swedish army. In 1980 I got selected to be an exchange student for my final 1.5 undergraduate years at the Swiss Federal Institute of Technology (ETH), Zurich. January 1982 I graduated as "Civilingenjör" (MS) from the LTH with final exams from the ETH. In March 1982 I obtained an employment at the Department of Automatic Control, ETH, Zurich as a teaching assistant. I attended the postgraduate courses in Automatic Control for three semesters, and since 1983 I have concentrated on the research project presented in this thesis. Since January 1984 I have been a research assistant with support in full from an ETH research grant. In 1984 I was appointed Lecturer in Simulation Techniques at the ETH.